POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

An association rule based framework for dynamic systems rebalancing: an application to bike sharing

Supervisor:
prof. Paolo Garza

Co-Supervisor:
prof. Sara Comai

Candidate:
Marco Cipriano
ID: s261380

Academic year 2019-2020

Contents

Chapter 1

Introduction

Bike sharing systems are spreading throughout the world, being used by more and more people every day. Anyway, a higher diffusion means more bikes to manage and reorganise among bike sharing stations to guarantee a good quality of the service. Up to today, it does not exist a universally accepted algorithm to rebalance the distribution of bikes over a dock-based bike sharing system.

From an extensive review of the current state of the art, it appears that, although bike sharing is a very active research field, there are few papers about dynamic rebalancing algorithms and all the existing ones are based on mathematical models of bike sharing stations.

This thesis presents a completely data-driven framework able to solve the dynamic rebalancing problem of a target system s, along with its application to Barcelona's bike sharing dock-based system.

The framework is able to find five different solutions to the rebalancing problem according to different time-based criteria and receives nine input parameters that can be modified at will to model and solve (almost) any type of rebalancing problem. For instance, two possible application domain are the dynamic relocation of employees in a chain of shops to address different flows of customers or the dynamic rebalancing of bikes inside a bike sharing system.

Nonetheless, for simplicity reasons, this thesis work will focus on the framework's application to bike sharing, offering only an high-level overview of the framework applied to abstract entities, i.e. to a general application domain.
In a nutshell, this thesis work will present a framework that receives Barcelona's bike sharing stations logs and positions as input and propose some bike movements in order to rebalance the system. The bike movements will depend on the input parameters of the framework and on the chosen time-based criterion that the

framework will exploit to gather information on the system's usage patterns.

This thesis work is organised as following:

The 2nd chapter contains information about related work: a description of theoretical and technical concepts used, a brief introduction to bike sharing and a review of the state of the art about bike sharing systems and rebalancing algorithms.

The 3rd chapter describes the input dataset and all the critical issues that need to be addressed in order to enhance the quality of the output.

The 4th chapter explains in detail every step performed by the framework to rebalance Barcelona's bike sharing system, along with the role of its input parameters.

The 5th chapter contains a discussion of the obtained results, comparing the performance of the framework for different values of its input parameters and rebalancing approaches.

The 6th chapter concludes the thesis, commenting the problems faced while realising the framework and presenting possible prompts for future work.

Chapter 2

State of the art

The purpose of this chapter is to give to the reader all the technical and contextual background needed to fully understand the framework presented in this thesis and the choices taken while developing it.

2.1 Theoretical concepts

The steps that allow to individuate and re-balance the critical elements of a system are typically based on large amount of *transactional* data produced by the system itself during daily operations. For this reason, it is necessary to rely on data mining techniques and concepts, such as *association rules* and the *FPGrowth* algorithm. Since these concepts may not be familiar to everyone, they will be described in the next sections.

2.1.1 Big Data, Data Science and Data mining

Nowadays, electronic devices are everywhere. Smartphones, smartwatches, computers, temperature sensors, IoT gadgets, are just examples of a myriad of devices that generate data, either on request of a final user or not. For instance, while you are using your smartphone to take a picture and post it on a social media, the car sharing automobile parked behind your house is sending its GPS position every minute to a server somewhere in the world.

All these devices generate flows of data with characteristics completely different with respect to the one existing in the past and which need innovative and scalable algorithms to extract knowledge from them.

These data are called **Big Data** and they can be defined as:

*"Data whose **scale**, **diversity** and **complexity** require new architectures, techniques, algorithms and analytics to manage it and extract **value** and **hidden knowledge** from it"* [1].

Their peculiar properties, already hinted in the aforementioned definition, can be described by the so-called 5Vs [2]:

- **Volume**: the amount of data to be analysed is very huge. For instance, in 2012 Facebook revealed that its system processed 2.5 billion pieces of content and 500+ terabytes of data each day [3].

- **Velocity**: in some conditions, these data can have a high generation rate, so it could be problematic analysing them in real time. For instance, traffic monitoring services (e.g. Google Maps and similar software) need to analyse large amounts of recent data in a very small time to offer a prediction of how busy a street is and to decide which is the best route to go from A to B.

- **Variety**: there are many different types of data: numerical, images, audio, video and so much more. Since they are in different formats, they need to be manipulated and merged to be used together.

- **Veracity**: the quality of data may not be high, so it must be heavily preprocessed and cleaned.

- **Value**: data contain meaningful insights in terms of money, but also businesses' advantage or information to improve a service.

The 5 Vs impose some new challenges both in terms of new techniques to manage and analyse these data and in terms of technology and infrastructures that should be modified to move the processing power to the data and not vice versa.

Data Science answers to these challenges through a combination of statistics, scientific methods, and data analysis to extract value from data. Data Science involves several different disciplines, but, for the purpose of this thesis, we are interested only to a small part of it, called **Data mining**.

Data mining is the process of automatically discovering useful information in large data repositories [4]. It comprehend several techniques, such as **association rules**, the one that is used in this thesis, classification and clustering.

2.1.2 Transactional data and association rules

An association rule is an implication expression of the form $X \rightarrow Y$*, where* X *and* Y *are disjoint itemsets, i.e.,* $X \bigcap Y = \emptyset$ [4]. It represents a frequent correlation or pattern contained in a transactional database, which can be seen as a list of transactions.

Therefore, to understand association rules and their metrics, it is necessary to take a little step back and try to understand what is a transaction.

A *transaction* is a set of non ordered values of any type related to a specific entity. If the elements are all of the same category then a transaction is also an *itemset*. For instance, an itemset could be the set of items contained in the receipt of a supermarket, such as:

Beer, *Bread*, *Milk*

Therefore an example of a transactional database could be:

#	Transaction
1	Beer, Bread, Milk
2	Diapers, Milk, Spaghetti, Water
3	Meat, Water
4	Diapers, Meat, Milk, Yogurt

Table 2.1. Example of transactional database

In the database presented in Table 2.1, all the rows are both transactions and itemsets. Nonetheless, {*Meat,Water*} or {*Diapers*} are valid itemsets contained in the database. To understand how widespread these two itemsets are inside a given database, it is possible to define two metrics called **support count** and **support**.

The **support count** of a given itemset i over a database db is defined as the number of transactions of db that contain i. For instance, the itemset {*Beer, Bread, Milk*} contains the itemset {*Bread,Milk*}, but does not contain the itemset {*Diapers, Milk*}. As a consequence, according to the database in Table 2.1, the support count of {*Meat,Water*} and of {*Diapers, Milk*} are respectively 1 and 2.

The **support** of a given itemset i over a database db is defined as the support count of i over the number of transactions of db. While the support count only gives an absolute number, the support is a relative measure that represents the

percentage of transactions of *db* that contain *i*. For this reason, the support is much more useful than the support count and will also be used for the association rules in the next paragraphs.
Pay attention that both support and support count benefit of the so-called *anti-monotone property of support*, for which if we drop out an item from an itemset, support value of new itemset generated will either be the same or will increase [5]. For instance, if an itemset {item1,item2} is contained in n transactions, then the itemset {item1} will be contained in all the n transactions, plus a variable number x of transactions that contain *item1* and not *item2*. Hence, the support count of the smaller itemset will be $n + x \geq n$.

Now that these concepts are clear, it is possible to explain what is an association rule, which information it contains and how it can be exploited.

Coming back to the initial definition, an **association** rule describes a frequent correlation or pattern contained in a transactional database and it is represented through two itemsets connected by an arrow. The first itemset is called *antecedent* or *body*, while the second is called *consequent* or *head* of the rule. An example of an association rule could be:

$$\{\text{item1, item2}\} \rightarrow \{\text{item3, item4}\}$$

where {item1, item2} is the *antecedent*, while {item3, item4} is the *consequent*.

The meaning of the rule is: *"If all the items of the antecedent are present in a given transaction, then, with a certain probability p, also the items of the consequent will be in the same transaction"*.

It is important to notice that p is not the probability that the presence of the antecedent in a given transaction *implies* the presence of the consequent, but is the probability of the *co-occurrence* of antecedent and consequent in the same transaction. The right interpretation of p is then: *"antecedent and consequent are both present in a given transaction t with probability p"*.

Association rules are associated to a list of interesting metrics useful to understand both the diffusion and the reliability of the association between the antecedent and the consequent. These metrics are **support** and **confidence**.

The **support** is probably the most important of the 2 considered metrics because it represents the aforementioned probability p. Given a transactional database *db*, the support of a rule r is computed as the support of the itemset containing both the antecedent and the consequent of r. This means that if r has a support equal to 0.7, its antecedent and its consequent are present together in 70% of the transactions contained in *db*. Therefore, if the transactions in the database are representative

of a given real world supermarket behavior [1], the rule will have a probability p of 0.7 to be found in a receipt produced by that supermarket.

For instance, using the list of transactions presented in Table 2.1, it is possible to extract at least 3 association rules and their supports.

#	Association rule	Support
1	Meat → Water	0.25
2	Diapers → Milk	0.5
3	Milk → Diapers	0.5

Table 2.2. Example of association rules with support extracted from Table 2.1

As it is possible to notice from Table 2.2, the support defines only how many transactions contain both the antecedent and the consequent of the rule, but does not provide any information about the support of either the antecedent or the consequent. If the antecedent and the consequent both have a very high support, a high support of the associated rule may not be meaningful. For instance, assuming that both antecedent and consequent of a rule r have a support of 0.8, r must have a support ≥ 0.6. In this specific case, support alone is not only inconsequential, but also misleading. Moreover, the support does not make any distinction between the antecedent and the consequent, so it loses the information contained in the direction of the rule itself.

For these reasons, it is needed a different metric that describes a rule's reliability, with respect to the direction of the rule itself. This metric is called **confidence**.

Confidence measures the reliability of the inference made by a rule. For a given rule r $X \rightarrow Y$, the confidence is computed as the support of r over the support of X. The higher the confidence, the more likely it is for Y to be present in transactions that contain X [4]. Computing the confidence of the rules in Table 2.2, it is possible to obtain Table 2.3.

It is then possible to observe that, even if the support of the 2nd and the 3rd rule is the same, the confidence discriminates between them. The 2nd rule has a confidence of 1.0, meaning that all the transactions that contain *Diapers* also

[1]Defining S as the dataset containing all the receipts produced by a given supermarket, we can define a representative set S^* as a special subset of an original dataset S, which satisfies three main characteristics: it is significantly smaller in size compared to the original dataset, it captures the most of information from the original dataset compared to any subset of the same size and it has low redundancy among the representatives it contains [6].

#	Association rule	Support	Confidence
1	Meat → Water	0.25	0.5
2	Diapers → Milk	0.5	1.0
3	Milk → Diapers	0.5	$0.\overline{6}$

Table 2.3. Example of association rules with confidence and support extracted from Table 2.1

contain *Milk*. The 3rd one, instead, does not have confidence 1.0, because not all the transactions that contain *Milk* also contain *Diapers*, as it is possible to see in the 1st transaction of Table 2.1. As a consequence, the most reliable rule among the 2nd and the 3rd one is the former, since the presence of its antecedent in an itemset *always* imply the presence of the consequent in the database.

2.1.3 Association rules extraction

Association rules extraction is performed through algorithms able to discover rules hidden in an input transactional database *db*. These algorithms always need two input parameters other than *db*: a minimum support *minSupp* and a minimum confidence *minConf*. These two parameters are needed to put some boundaries on the rules that the algorithm will extract, enabling it to converge in a limited time. If *minSupp*, *minConf* or both are too low, the association rules extraction will request too much time or memory and will not converge[2].

A common strategy adopted by many association rule mining algorithms is to decompose the problem into two major sub-tasks [4]:

1. **Frequent Itemset Generation**, whose objective is to find all the itemsets that satisfy the *minSupp* threshold. These itemsets are called *frequent itemsets*.

2. **Rule Generation**, whose objective is to extract all the rules with confidence higher than *minConf* from the frequent itemsets found at the previous step.

[2]Practical example: a real supermarket could sell quite easily 100 distinct items in a month. Assuming the minimum support and the minimum confidence to be 0, the number of possible rules that can be extracted from the dataset containing all the receipts produced by the aforementioned supermarket in one month is $3^{100} - 2^{100+1} + 1 \approx 5 \cdot 10^{47}$ [4], too much for any computer or cluster. The same example holds for value of minimum support and minimum confidence different from 0, but still too low to decrease enough the number of generated rules.

Frequent itemset generations is the most computationally expensive step among the two, so typically the algorithms focus on performing this first phase efficiently, more than the second one.

In this thesis, it was decided to use the FP-Growth algorithm, since it was already implemented and parallelised in the MLlib library [7] on Apache Spark[3], the chosen framework.

In broad terms, the **FP-Growth** algorithm operate as following: given a list of transactions, the first step of FP-growth is to calculate item frequencies and identify frequent items. Different from Apriori-like algorithms designed for the same purpose, the second step of FP-growth uses a suffix tree (FP-tree) structure to encode transactions without generating candidate sets explicitly, which are usually expensive to generate. After the second step, the frequent itemsets can be extracted from the FP-tree [9]. A more in depth explanation of the algorithm can be found at [4], while the papers related to the sequential and parallel version of the algorithm are respectively [10] and [11]. The MLlib library implements the parallel version.

Anyway, to understand the following chapters, it is only needed to know that the FP-Growth algorithm receives as inputs a minimum support *minSupp*, a minimum confidence *minConf* and list of transactions. As output, it produces all the itemsets having a support greater or equal to *minSupp* and all the rules having respectively support and confidence greater or equal to *minSupp* and *minConf*.

2.2 Application domain analysis

The application domain chosen to show the capabilities of the presented framework is bike sharing, due to the related re-balancing problem. There is plenty of recent papers that try to solve it, so it is a very active field, open to new approaches and algorithms. However, to understand why the aforementioned re-balancing problem occurs, it is necessary to be familiar with bike sharing systems and the basic concepts behind them.

Bikes and stations are the most important physical elements of any bike sharing system. Bikes are quite standard in size, color, and configuration. Stations, instead, are modified bike racks able to securely lock parked bikes. In some cases, as the one considered in this thesis, stations are also able to log the number of taken and free slots every few minutes, sending this information to a server or keeping it locally.

[3]Apache Spark is a unified analytics engine for large-scale data processing. It provides high-level APIs and an optimized engine that supports general execution graphs [8].

Each station has a number of docks between 10 and 100 or more, depending on local traffic volumes. Users are able to take bikes from the stations through credit cards or electronic keys, depending on the system itself. When they do not need anymore a bike, they have to park it in one of the stations of the bike sharing system.

Since users are not forced to park the bike in the station in which they took it, the system is often unbalanced, having some or, in the worst cases, the majority of the stations completely empty or full. This phenomenon causes disruptions to the bike sharing service, making difficult for the users either to find a parked bike or to park the one they are currently using. For this reason, typically bike sharing employees move bikes across the stations by truck or trailer daily or weekly. The set of movements that the employees perform depend on the rebalancing algorithm the bike sharing company decide to use.

2.3 Related work

As mentioned in Section 2.2, bike sharing is a very active research field. It is possible to divide the research papers in 3 main areas, analysed in the following subsections:

- **Stations behavior analysis**: papers which study the behavior of the bike sharing stations, determining which factors influence the usage of a given station and, in some cases, predicting their behavior in the future.

- **Static rebalancing**: papers which present algorithms to rebalance the bike sharing system making the *static* assumption , i.e. they assume the system to be isolated from end users when the rebalancing takes place. Static approaches must be performed during the night to avoid being too troublesome for the final users, that can not interact with the service in that period. As a consequence, static algorithms can guarantee the system to be perfectly balanced in the morning, but can not avoid failures due to heterogeneous usage of the stations during the day.

- **Dynamic rebalancing**: papers that present algorithms which do not need the *static* constraint to work properly. They support continuous operations of the system, taking into account its real-time usage and updating the redistribution strategy as soon as new information are revealed.

2.3.1 Stations behavior analysis

Faghih-Imani et al. [12] studied the factors that drive usage of bike sharing systems and developed a mixed linear model to estimate the influence of bicycle infrastructure, sociodemographic characteristics and land-use characteristics on customer arrivals and departures to and from bike sharing stations. Further, they developed a binary logit model to identify rebalancing time periods and a regression model framework to estimate the amount of rebalancing. Similarly, Hulot et al. [13] focused on predicting the hourly demand for demand rentals and returns at each station of the system. Their model extracts main behaviors of stations and combines them to build a specific model for each station. Fricker et al. [14] proposed a stochastic model of an homogeneous bike-sharing system and studied the effect of the randomness of user choices on the number of "problematic" stations, i.e. stations that, at a given time, have no bikes available or no available spots for bikes to be returned to. They quantified the influence of the station capacities and found that the fleet size optimal in terms of minimizing the proportion of problematic stations is half of the total number of spots plus a few more. The value of the few more can be computed in closed-form as a function of the system parameters. Moreover, they found that simple incentives, such as suggesting users to return to the least loaded station among two stations, improve the situation by an exponential factor.

2.3.2 Static rebalancing

Both Chemla et al. [15] and Cruz et al. [16] proposed efficient algorithms to solve the bike sharing rebalancing problem statically. Their algorithms assume that the rebalancing is performed through a single vehicle with limited capacity, but this assumption holds only if the target city is divided in districts with one truck each. In all the other cases, that may lead to more performing solutions, the algorithms presented in these papers can not be used. Dell'Amico et al. [17] presented four mixed integer linear programming formulations of the rebalancing problem. They removed the aforementioned single-vehicle assumption, employing a fleet of capacitated vehicles to re-distribute the bikes with the objective of minimizing total cost.

2.3.3 Dynamic rebalancing

Chiariotti et al. [18] proposed a dynamic rebalancing strategy that exploits historical data to predict the network conditions and promptly act in case of necessity. They used Birth-Death Processes to model the stations' occupancy and to decide when to redistribute bikes, and graph theory to select the rebalancing path and the

stations involved. They found their dynamic approach to rebalancing to outperform static ones, while allowing system controllers to decide whether to prioritize maintenance costs or service quality.

2.3.4 Chosen approach

The framework presented in this thesis implements an algorithm to solve the dynamic rebalancing problem. There are multiple reasons behind the decision to work on a dynamic algorithm rather than on a static one, such as:

- There is very limited literature on dynamic algorithms.
- Dynamic rebalancing is more versatile than static one.
- Dynamic rebalancing outperforms static approaches, especially if it is decided to rebalance the system multiple times per day.

Furthermore, it was decided to adopt a completely data-driven approach, because, to the best of my knowledge, all the other static and dynamic algorithms are based on statistic approaches and mathematical models of bike sharing systems. While these algorithms can be more or less effective according to the assumptions they are based on, a full data-driven approach should perfectly fit the problem and discover patterns hidden in the data themselves. Moreover, being based only on the data produced by the system on which it will operate, the proposed framework should work optimally with any bike sharing system, independently of the system parameters such as the number of stations or the fleet size.

Chapter 3

Input Dataset analysis

3.1 Dataset description

The selected input dataset contains two input sources. The former, called Stations.csv[1] provides information about the bike sharing stations in Barcelona. It contains 3301 stations and for each of them it defines a unique positive identifier called *station id*, along with the station name, longitude and latitude. For instance, Table 3.1 displays the first 5 rows of this file.

The latter, called Register.csv, is a log file produced by Barcelona's bike sharing system between May and September 2015. It contains $\approx 2.5 \cdot 10^7$ structured entries, organised on four columns, as shown in Table 3.2:

- **Station id**: identifier of the station to which the line is referred. It can be seen as an external key to *stations.csv*.
- **Timestamp**: the timestamp t in which that line was produced. A correctly working station is supposed to generate a log line every two minutes.
- **Taken slots**: number of bikes parked in station *station id* at timestamp t.
- **Free slots**: number of free slots of station *station id* at timestamp t.

[1]There are various specifications and implementations for the CSV (Comma Separated Values) format. In the context of this thesis, a csv file is a tabular file having an header containing the name of the columns separated by commas as first row, while the following rows contain the actual file values. Additional information on CSV format are available in RFC 4180 [19].

row	Station id	Longitude	Latitude	Station name
1st	1	2.180019	41.397978	G.V. Corts Catalanes
2nd	2	2.176414	41.394381	Plaza Tetuán
3rd	3	2.181164	41.393750	Ali Bei
4th	4	2.181400	41.393364	Ribes
5th	5	2.180214	41.391072	Pg Lluís Companys

Table 3.1. First rows of *stations.csv*

row	Station id	Timestamp	Taken slots	Free slots
1st	1	2008-05-15 12:01:00	0	18
2nd	1	2008-05-15 12:02:00	0	18
3rd	1	2008-05-15 12:04:00	0	18
4th	1	2008-05-15 12:06:00	0	18
5th	1	2008-05-15 12:08:00	0	18

Table 3.2. First rows of *register.csv*

3.2 Dataset critical issues

The dataset presents some critical issues that the framework needs to address to perform an effective rebalance:

- Some log lines represent stations having both zero taken and zero free slots. This behavior could be either generated on purpose by an out of order station, for instance for maintenance, or can be due to local malfunctioning. Whichever is the right interpretation, this phenomenon has to be taken into account.
- Some log lines, instead, are related to stations with just one slot, having either one free slot and zero taken or vice versa. It is reasonable to assume that, if a given station have just one working slot because the others are in maintenance, that station is completely disabled. Hence, it is likely that these lines are related to malfunctioning stations more than to real configurations. Differently, it was assumed that stations with two or more working slots would still be beneficial for end users, so they are considered correctly functioning and are not mentioned in the critical issues.
- Not all the station in *Stations.csv* generate log lines on *Register.csv*. The

stations present in at least one log line will be called *log capable* stations. This issue could occur for two reasons: either some of the stations present in *stations.csv* are not able to log their occupancy or the data generated by them are intentionally not included in *Register.csv*. Anyway, it is not possible to make any prior assumption about the number of log rows produced by an arbitrary station s.

- Some timestamps do not have a log line for each *log capable* station. As a consequence, the status of some stations is unknown in some of the timestamps. Furthermore, some log lines are also unaligned, i.e. are not generated every two minutes, as displayed in the 1st row of Table 3.2.

- The number of slots of a given station, computed as $Taken\ slots + Free\ slots$, can be subject to heavy fluctuations over time.

Some examples of these issues are represented in Figures 3.1, 3.2 and 3.3. In these charts the x axis represent a timestamp identifier, so a growing unique number introduced to speed up the figure generation while preserving the data order, while the y axis represents the number of slots of the target station per timestamp before data cleaning.

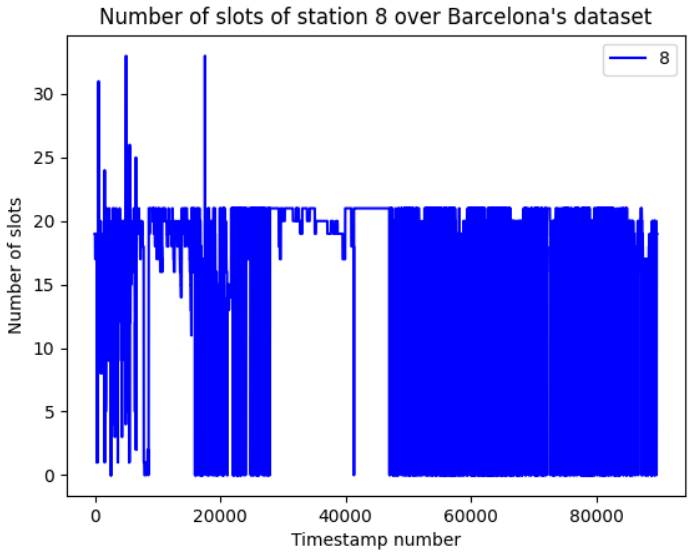

Figure 3.1. Number of slots of station 8 over Barcelona's dataset. It is possible to notice that the number of slots is frequently equal to 0, probably because the station is often unavailable. This behaviour must be addressed while preprocessing the data.

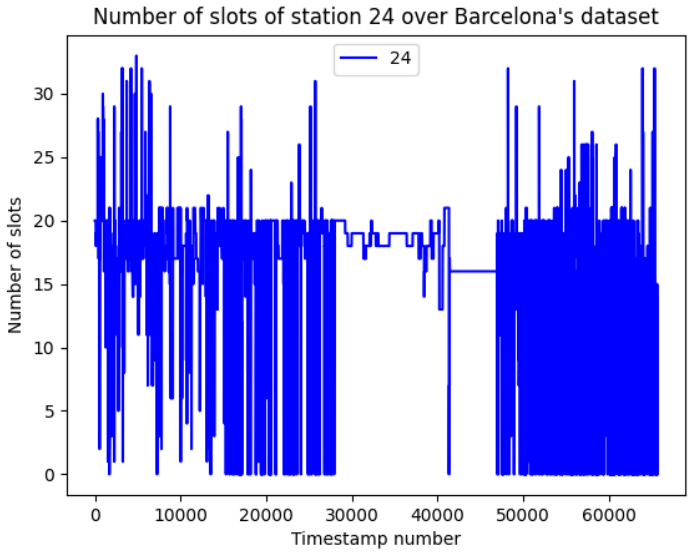

Figure 3.2. Number of slots of station 24 over time. It is possible to observe that the number of timestamps is much lower than on the other two charts. As a consequence, considering the whole dataset, the status of this station is frequently unknown.

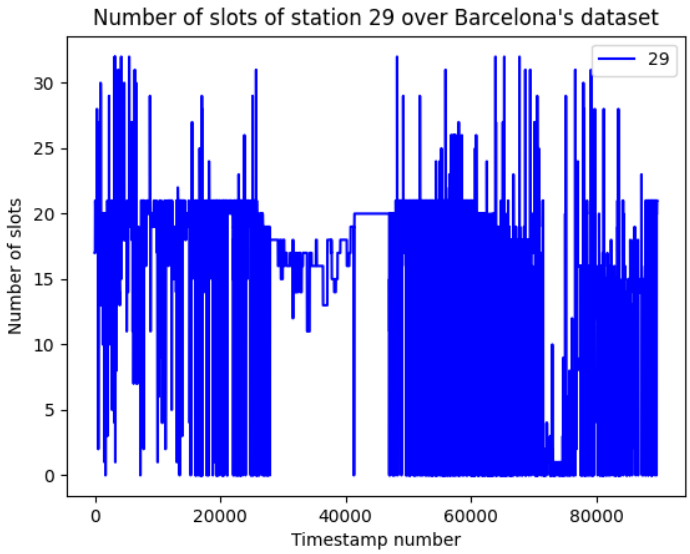

Figure 3.3. Number of slots of station 29 over time. This station have a number of slots that is often equal to 0 and is also subject to heavy oscillations over time.

Chapter 4

Proposed framework

The proposed framework is developed in Python and Java. It was decided to use Python in order to build a versatile and completely automated system, while Java was exploited for the high execution parallelism provided by Apache Spark JavaRDD approach[1] for the most computationally intensive tasks.

The framework was designed to be highly versatile and applicable to a large set of use cases, therefore this chapter will firstly provide an overview of the framework applied to a system of abstract entities, in order to give an idea of how the framework would rebalance a generic system.

Subsequently, to present the framework's real-world application to bike sharing, these abstract entities will be one-to-one mapped to concrete ones in the application domain and the core concepts and definitions on which this specific application is based will be formalised.

Finally all the framework blocks and sub-blocks will be analysed in detail, discussing the choices taken while designing it, along with the role of the input parameters affecting its execution.

The results produced by the framework, divided per logic block, will be discussed in Chapter 5.

[1]Spark revolves around the concept of a Resilient Distributed Dataset (RDD), which is a fault-tolerant collection of elements that can be operated on in parallel. Additional information can be found at https://spark.apache.org/docs/3.0.1/rdd-programming-guide.html#resilient-distributed-datasets-rdds [20].

4.1 General framework overview

The framework has two input sources:

- a log containing historical data about the usage of the system to be rebalanced.
- a file containing the position (longitude and latitude) of all the elements of the system.

In order to be able to measure the performance of the framework, the first step is to divide the input dataset in two different sets: training and validation set. Subsequently, both these sets are partitioned according to different time-based criteria to generate four sets of data of different granularity.

For each timestamp in each partition of the training set, the framework identifies the *critical* elements, according to a function *f(e, near_elements)*, where e is the potentially critical element, while *near_elements* are the elements distant at most d meters (air-line distance) from e, with d being a parameter of the framework. These *near_elements* are also called *neighbourhood* of e.

From this data, the framework extracts association rules, to understand which elements are frequently critical together. Finally, association rules having the antecedent exclusively composed by elements belonging to the neighbourhood of the consequent are isolated and used to perform corrective actions (applied) on the validation set in different moments of the day, called *repositioning hours*.

The framework allows to evaluate the performance of the system rebalancing computing the number of critical elements before and after rules application on the validation set. Furthermore, it outputs the number of requested corrective actions, in order to understand the impact of a single action on the system behavior and the effectiveness of the extracted rules.

This overview highlights the extreme versatility of the rebalancing framework, which can be used in any type of continuous supply and rebalance of goods, such as the distribution of items in a supermarket chain or to optimise the distribution of bikes in a bike sharing service. To adapt the framework to a given application domain, it is only needed to map the aforementioned abstract entities to concrete ones.

For the reasons explained in the introduction and in Section 2.3.4, this thesis work will focus on rebalancing bike sharing systems. Therefore, the mapping for this specific application domain is the following:

- **Element**: bike sharing station.

- **Function *f(e, near_elements)***: an element e is critical if the percentage of taken slots over the total number of slots (*occupancy rate*) is higher or lower than the average *occupancy rate* of its neighbourhood by a threshold $t\%$, with t as a parameter of the framework. Actually, the concept of "critical" station is more complex than it seems and it is at the core of the presented framework, hence more details about it will be provided in the following section.
- **Corrective action**: movement of b bikes from station x to station y.
- **Training and validation sets**: the validation set is defined as the last 7 days of each month, while the training set contains the remaining days.

Since the literature around rebalancing algorithms for dock-based stations is very limited, this application domain is a very interesting testing field for the presented framework. Therefore, the following chapters will focus on the application of the proposed framework on the bike sharing application domain. This decision has two main advantages: it enables this thesis work to provide a more detailed description of the steps that the framework carries out to rebalance the target system and gives the possibility to show the framework's performance while solving a real-world rebalancing problem, instead of a fictional one.

When applied to this specific application domain, the framework's behaviour strongly depends on some fundamental definitions, such as the *neighbourhood* and *critical station* ones. Therefore, before digging deeper in the actual framework implementation, it is essential to analyse and fully understand these core concepts and how they affect the proposed system rebalancing.

4.2 Core concepts and definition

All the following concepts are used by the framework to discriminate between situations that need corrective actions from the employees and "normal" ones. Any change in them would completely change the number and the features of the "critical" elements, resulting in the extraction of completely different patterns.
As a consequence, these definitions are considered the real core of the framework.

The definition of "critical" stations is based on a measure called *occupancy rate* and on the meaning of the *neighbourhood* of a station, so the first step is to give a formal definition of these two concepts.

Defining S as the set of stations of the bike sharing system to be rebalanced, given a station $s \in S$ having a number of taken slots $taken\_slots_s$ and a number of free slots $free\_slots_s$, the occupancy rate $Occupancy\_rate$ of s is defined as:

$$Occupancy\_rate(s) = \frac{taken\_slots_s}{(taken\_slots_s + free\_slots_s)} \tag{4.1}$$

Furthermore, the neighbourhood of s can be defined as:

$$Neighbourhood(s) = \forall s_i \in S \mid distance(s, s_i) < d \tag{4.2}$$

Where $distance(x, y)$ is the air-line distance between station x and y, while d is a positive parameter of the framework. Furthermore, since the neighbours of s are the stations inside a circle of radius d from s, s will be called *center* of its neighbourhood. Finally, pay attention that since $s \in S$ and $distance(s, s) = 0 < d$, the station s is included in its own neighbourhood.

It is then possible to define a "critical" station as a station s for which holds:

$$\mid Occupancy\_rate(s) - \frac{\sum_{i=1}^{n} Occupancy\_rate(s_i)}{n} \mid > t \tag{4.3}$$

Where s_i is a generic stations among the n station in the neighbourhood of s and t is a decimal value for which holds $0 < t < 1$, received as a parameter of the framework.

In simple terms, a station is "critical" if its occupancy rate is higher or lower than the average occupancy rate of its neighbourhood of at least a threshold t.

Anyway, this definition does not make any distinction between stations that have an occupancy rate higher than the neighbour average and stations that, instead, have an occupancy rate that is lower than the average. Since distinguishing between

bike stations that usually have too many bikes from ones that, instead, are usually almost empty could be really useful to perform bike rebalancing, it was decided to extend the definition at 4.3, preserving the sign.

Therefore, a station s is "positively critical" if it holds:

$$Occupancy\_rate(s) - \frac{\sum_{i=1}^{n} Occupancy\_rate(s_i)}{n} > t \tag{4.4}$$

while it is defined "negatively critical" if it holds:

$$Occupancy\_rate(s) - \frac{\sum_{i=1}^{n} Occupancy\_rate(s_i)}{n} < -t \tag{4.5}$$

As a consequence, a "positively critical" station will have a higher occupancy rate than its neighbourhood's average, while a "negatively critical station" will have a lower occupancy rate than its neighbourhood's average. For simplicity reasons "positively critical" and "negatively critical" stations will be respectively called "positive" or "negative" stations where needed. Furthermore, the type of criticality will also be called "sign". For instance, a set of positively critical stations will have a different "sign" with respect to a set of negatively critical stations.

Pay attention that both the "occupancy rate" and the "critical" definition for a station s depend on the considered timestamp, for which the number of taken and free slots of s is defined. Hence, a station that is positively critical in a timestamp t_1, may be negatively critical or non critical at all in a timestamp $t_2 \neq t_1$.

It is possible to notice that it was decided to use both the occupancy rate and the neighbourhood concepts as cornerstones of the "critical" definition to individuate problematic configurations in the bike sharing system.
The idea that led to this choice is that, on average, the occupancy rate of a station should be both similar and related to the one of the other stations nearby. If a station has an occupancy rate much higher or much lower than its neighbourhood, that station is strongly[2] under or overused. As a consequence, corrective actions are needed to avoid that station to become completely full or empty, hence potentially useless for end users.

The two core parameters of these definitions are the air-line distance d and the threshold t. High values of d determine large neighbourhoods, more difficult to rebalance but allowing to observe complex correlations in behaviour of distant

[2]The strength of this statement depend on the critical threshold t. The higher t, the higher the difference in usage.

stations. Low values, instead, generate smaller station clusters, for which it is easier both to explain the usage patterns and to reallocate bikes, but that could make the framework lose important insights on the system behaviour.

The value of t, instead, distinguish between what the framework considers "abnormal" and what "normal". Low values of t highlight both small and high variations in occupancy rate inside a neighbourhood, but can hide the most interesting patterns in the data. For instance, setting the critical threshold t to 0.05 would make the framework signal as critical the large majority of the stations for each input timestamp t, hiding the useful patterns in a mountain of meaningless ones. On the other hand, high values of t lead to a small number of critical stations, but can ignore even more important problems. Namely, a 50% difference in occupancy rate between a station and its neighbourhood is indeed important, but if it happens just once per week it can be ignored. By contrast, a 20% difference that is present every morning at 9:00 must be dealt with.

4.3 Framework structure

Once that all the core concepts are defined and clear, it is possible to analyse the framework structure in depth. The framework proposed to dynamically rebalance Barcelona's bike sharing system can be decomposed in 3 main blocks:

- **Preprocessing**: it addresses all the input dataset critical issues presented in Section 3.2, cleaning as much as possible the input data. Furthermore, it divides the dataset in four time-based partitions. The reasons behind this partitioning approach will be analysed in depth in Section 4.4.2.
  Preprocessing is the foundation of the whole framework, since on its effectiveness depends both the quality of the rules extracted in the following block and the overall framework performance.
- **Rules extraction**: it elaborates the data generated by the preprocessing block in order to extract association rules between stations that are frequently critical together. This block is the intellect of the framework, since it is able to find patterns in the usage of bike sharing stations, generating valuable knowledge that will be used to rebalance the bike sharing system.
- **Station rebalancing**: it exploits the insights gathered in the previous block, producing the instructions for the bike sharing employees to rebalance the system. Moreover, it measures the framework performance applying the proposed bike movements to the bike sharing system and comparing the number of critical stations before and after the changes.

4.4 Preprocessing

The preprocessing block receives as input the dataset described in Chapter 3, addresses all the individuated critical issues and generates a partitioned dataset that can be easily exploited to extract the association rules in the rules extraction block. It consists of two distinct parts: the former is a Java algorithm which exploits Apache Spark [8] to achieve high-performance data cleaning and re-elaborates the information on stations position for the following blocks, while the latter is a Python algorithm in charge of dividing the already cleaned data in different partitions.

4.4.1 Data cleaning and restructuring

The quality of any algorithm output is strictly linked to the quality of input data. For this reason, an effective data cleaning algorithm is fundamental in any data mining based framework.
The first step is then to analyse all the critical issues found in the input dataset and individuate corrective actions for each of them.

- **Log lines with 0 taken and 0 free slots**: these lines do not contain general information about the usage of the bike sharing system, other than reporting a station malfunctioning or maintenance. Since the purpose of this thesis is not to measure the quality of Barcelona's bike sharing system, these lines are considered outliers and **removed from the input dataset**.

- **Log lines with taken + free slots = 1**: this issue can be considered as an extension of the previous one, so it is subject to the same considerations. Hence, the chosen corrective action is to **remove these lines from the input dataset**.

- **Some stations in *Stations.csv* do not generate any log**: in this case the file affected is *Stations.csv* more than the log itself. As a consequence, the problem in this case is more related to the coherence of the output than on the information that can or can not be retrieved from the log. For this reason, it was decided to **filter all the stations not present in *Register.csv* at the end of the preprocessing phase**.

- **Some stations do not log their occupancy for each timestamp in *Register.csv***: to extract meaningful association rules, data should describe reliably the behavior of the bike sharing stations inside the system. For this reason, the more information are available for a given station, the better. In this case, it was decided to remove all the stations, and related log lines,

that are not present in at least a threshold $f\%$ of the timestamps present in *Register.csv*, where f is a parameter of the framework. The stations respecting this criterion will be called *frequent* stations.

- **The number of slots of bike sharing stations fluctuates a lot over time**: while the previously discussed issues were either related to single timestamps or on the presence/absence of an information, this issue depends on the evolution of a value over time.
  The statistical metric that best describes the strength of variations of a value over time is the *Variance*, computed as:

$$Variance(x) = \sigma^2(x) = \sum_{i=1}^{n}(x_i - \overline{x})^2 \tag{4.6}$$

  where $\overline{x}$ is the average value of x over the considered values. The higher the variance, the stronger the fluctuations.
  In the considered case, x represents the total number of slots of a given station s, x_i is the number of slots of s at timestamp i and $\overline{x}$ is the average number of slots of station s over the whole dataset. Defined this metric, it was decided to remove all the stations having $Variance > maxVariance$ in order to have reasonably stable stations while being still able to exploit a large dataset, where $maxVariance$ is a parameter of the framework. Therefore, stations respecting the $maxVariance$ criterion will be called *stable* stations.

To work on homogeneous data, the last cleaning step is to remove all the timestamps that do not contain all the stations survived to the previous filters.
In this way, it is possible to consider the same number of log lines for each station, while knowing their status in all the considered timestamps.

Anyway, there is one more step to be performed in this part of the framework, because all the next blocks need to know which are the neighbours of each station. Since this is a static information solely subject to the position of the stations and the value of the input parameter d, it was decided to compute the neighbourhood of each *frequent* and *stable* station during the preprocessing phase, storing them in a file called *nearStations.txt*. Each row of this file contains a station id, separated by a dash from all its neighbours, as displayed in Table 4.1.

Since all the presented steps need to operate on large amount of data, it was chosen to write this part of the algorithm in Java. Using this approach, it was possible to clean the input data in less than 5 minutes on a cluster of 16 nodes.

The Java program also generates some statistics on the input data, saving them in different files. These files will not be used by the next blocks, but can be

row	Station id - Neighbours
1st	7 - 8,10
2nd	8 - 7,10
3rd	10 - 7,8,40,41,42,43
4th	12 - 13,14,120
5th	13 - 12,14,16,49,120

Table 4.1. First rows of *nearStations.txt*

accessed by bike sharing system administrators to retrieve some statistics on the preprocessing block and to check that everything is working properly. In detail:

- **driverLogs.txt**: contains the number of rows of each input file both before and after the data cleaning, along with the number of rows that were filtered by each of the aforementioned criterion. This file will be used in Section 5.1 to show the actual performance of the preprocessing block.
- **stationsDistribution.txt**: for each station s, it contains the percentage of timestamps in which their status is defined, i.e. the number of log lines produced by s over the total number of timestamps. It is generated right before removing the not *frequent* stations and was used to choose a starting value for the minimum frequency f. It can still be used by an administrator to understand which stations are usually unavailable in its bike sharing system.
- **meanAndVariance.txt**: contains mean and variance for each of the *frequent* stations. It was used to determine a starting value for $maxVariance$, but can still be used by an administrator to understand which stations usually work properly and which need frequent maintenance.
- **Stations.csv**: it is the filtered version of the input file *Stations.csv*, containing only the stations still present in *Register.csv*.

Moving to the program structure, it was decided to represent it through a graph (Figure 4.1) to make it easier to understand. Each oval node represents a JavaRDD, while squared ones represent JavaRDDs that are also program outputs. As it is possible to notice, there is one squared node for each output file described before besides *driverLogs.txt*, since this specific file is generated as a list line by line during the execution of the program and is stored as output as last step. Hence, it is not a JavaRDD.

The output is a dataset modeling a perfect system from a structural point of view, i.e. for each timestamp it is possible to retrieve the status of all the stations. As a consequence, the next blocks are not affected by the quality of the logging framework of Barcelona's bike sharing system. This first block makes the framework easily generalisable and applicable to any city, independently of the quality of the data generated by the target system.

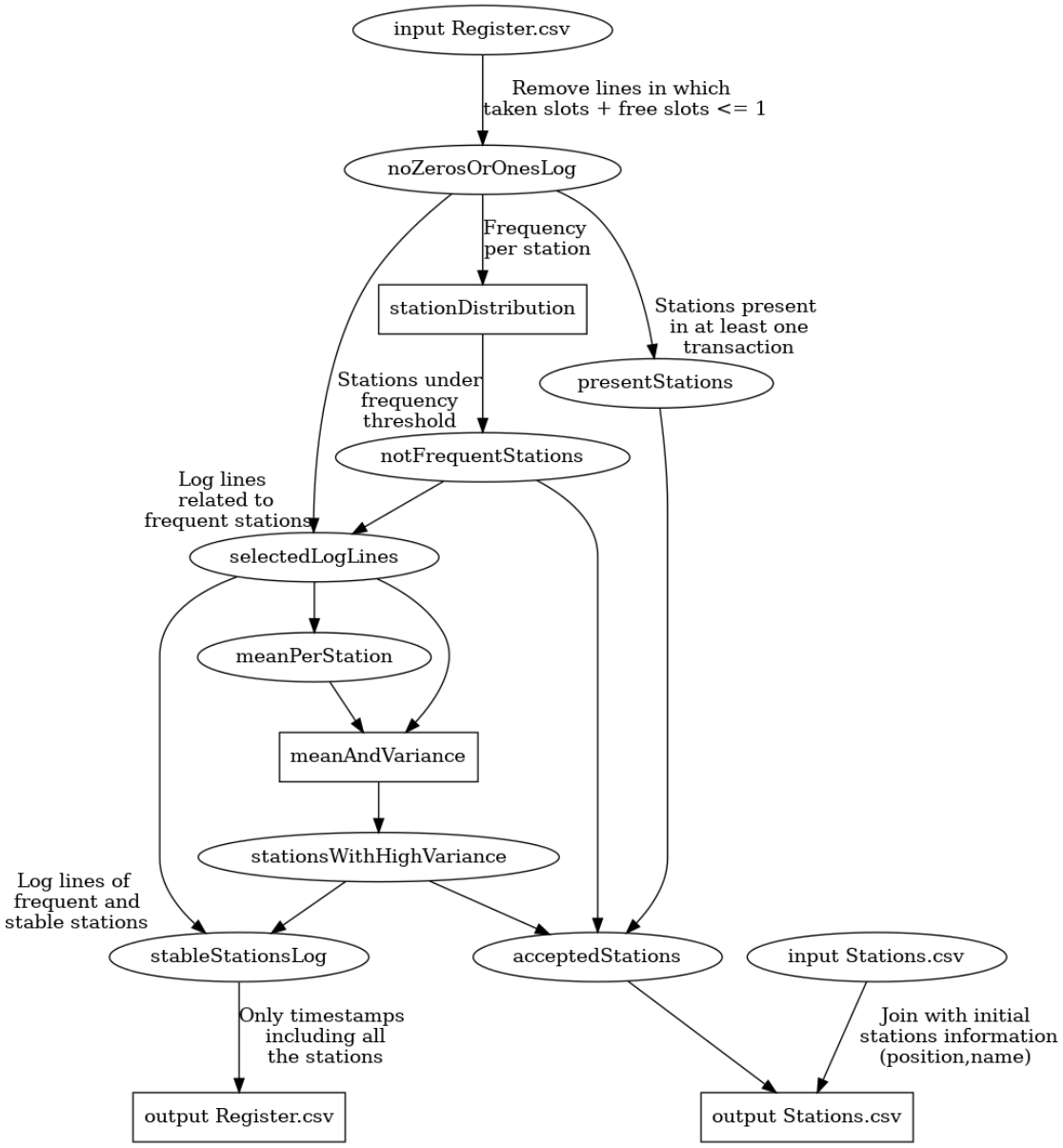

Figure 4.1. *Data cleaning and restructuring* JavaRDDs structure.

4.4.2 Data partitioning

The objective of this phase is to partition the input dataset to enable the next blocks to extract insights from the data at different levels of time granularity and to provide 5 different rebalancing strategies for the target system. In this way, it will be possible to compare the performance of the different rebalancing approaches and select the one that best fits the target system.

Since the tasks performed by this part of the framework do not need a high degree of parallelisation, they are performed in Python.
Pay attention that all the following steps will be applied month by month on the input dataset.

The first step is to divide the dataset in training and validation set. The training set will be used to learn the usage patterns of end users in Barcelona, while the validation set is needed to measure the performance of the framework without being biased. These two sets need to be disjoint from each other, so the framework can verify if the patterns extracted from the training set are also present in the validation set. If that is true, then it is highly probable that these patterns will be present also on new data. This approach is very common both in data mining and machine learning [21].
It was decided to have a validation set composed of the last 7 days of each month, while the training set contained the other days. This decision was taken in order to have a complete week in the validation set, so the framework would be able to verify the presence of patterns related to different days of the week, for instance present on Monday and not on Wednesday.

Since Barcelona's bike sharing system usage patterns are not known a priori, it was decided to try different approaches instead of performing assumptions. For this reason, the next step is to partition both training and validation set according to different time-based criteria to generate four sets of data of different granularity:

- **month partitioning**: all monthly data are kept together. This partition is useful to extract patterns that are always present during the month, such as "station 14 and station 20 are always critical together", for instance because they are both positioned near a metro station and are almost always empty or full. This partitioning strategy will lead to two different rebalancing approaches that will be described in Section 4.6.3.

- **per day partitioning**: 7 different partitions, one per day of the week. This partition should be able to capture usage patterns correlated to the day of the week. For instance, using these partitions, the framework should be able to gather insights on stations that are critical together every Monday.

- **per time slot partitioning**: n different partitions, according to the number of the chosen time slots, with n as a parameter of the framework. For instance, 3 possible time slots are: 5:00 to 13:00, 13:00 to 21:00, 21:00 to 5:00. This approach is orthogonal to the previous one, enabling the framework to gather usage patterns related to the time of the day, i.e. morning, afternoon or evening.
- **per day and per time slot partitioning**: it is a combination of 2$^{\text{nd}}$ and 3$^{\text{rd}}$ partitioning approaches. Each day of the week is divided in n partitions, generating $7 \cdot n$ partitions overall. It is the most specific partitioning and should be able to collect very specific behaviours, such as people going for a ride on Sunday morning.

These partitions will be used to perform four different analysis. In this way, the framework will observe the data at different levels, capturing many different usage patterns and exploiting each of them to perform the rebalancing.

As an additional advantage of this approach, since the framework does not leverage on any behavioural assumption models and is solely based on the log data, it does not need to be modified to be used in different cities, even if the behaviour of the target city's end users is completely different with respect to Barcelona's ones.

4.5 Rules extraction

Once the preprocessing is finished, the rules extraction block is in charge of finding the usage patterns for each of the defined partitions in terms of association rules. Since this block is the core of the whole presented framework, it is useful to dig a bit deeper in order to understand which patterns are being extracted and how they can be used in this phase.

Association rules extraction is a very computationally demanding operation, so it is carried out in Java, exploiting the Apache Spark framework [8].
The Java algorithm can be divided in two sub-blocks: **Critical stations detection** and **Association rules extraction**. Once the rules are extracted, the framework computes some statistics during a **postprocessing** phase implemented in Python.

4.5.1 Critical stations detection

As stated before, the Java algorithm for rules extraction can be divided in two parts. Since this sub-block is the first one, it will receive the output of the preprocessing phase as input. In detail, the two input sources of this section are the cleaned log file and the list of neighbours per station.
Starting from these two files, this sub-block is able to detect, for each timestamp contained in the input log, the list of stations that are in a positively or negatively critical situation.

The program structure is represented in Figure 4.2 at page 35. As in Figure 4.1, the oval nodes are RDDs, while the squared ones are also outputs. It is strongly recommended to look at the schema along with the following explanation to fully understand the main steps of the program.
The key aspects and transformation to which the input is subject are:

1. Input files are read from the disk and moved into the distributed main memory. They are called *fileLog* and *fileNearStations*.

2. It is computed the occupancy rate for each input stations, joining it with the near station data. From a logical point of view, the idea is to produce a list for each couple station $s_i \in S$ and timestamp $t_j \in T$ containing the occupancy rate computed in t_j of all the stations belonging to $Neighbourhood(s_i)$, where S is the set containing all *frequent* and *stable* stations, while T is the set of timestamps composed by all the timestamps present in the cleaned log. This

data structure[3] is called *nearStationsLog*.

3. The next step is to compute the average occupancy rate per neighbourhood for each timestamp $t_j \in T$, in order to compare this value with the occupancy rate of the respective centers.
   For instance, if s_1 is the center of the neighbourhood composed by the stations s_1, s_3, s_5, then, for each timestamp $t_j \in T$, this step of the algorithm will compare the occupancy rate of s_1 with the average occupancy rate of s_1, s_3 and s_5, in order to determine if s_1 is critical.
   This process generates a list called *transactions* containing one entry for each $t_j \in T$. Each entry contains all the critical stations in t_j in terms of station identifiers, distinguishing between positively and negatively critical stations using the sign of their identifier. As mentioned in Section 3.1, the station identifiers are all positive numbers, so it was decided to identify positively critical stations with a positive integer and negatively critical ones with a negative integer. This list is also the final output of the sub-block.

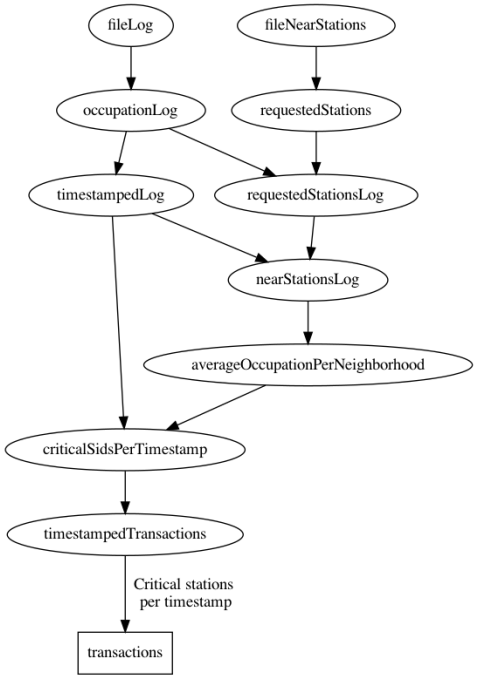

Figure 4.2. critical stations detection JavaRDDs structure.

[3]The actual implementation is an HashMap<String,List<Double>> having the station identifier concatenated to the timestamp as key and the list of occupancy rates of the stations' in its neighbourhood as value. Pay attention, $Neighbourhood(s_i) \ni s_i$.

This sub-block can be also used as a standalone program, since it has a command-line input argument that allow the framework to execute it alone, without the rules extraction part. In fact, the station rebalancing block will leverage on this sub-block to compute the critical stations both before and after the rebalancing. In that case, the output is not only a list of critical stations, but contains also the timestamps in which these stations are critical.
Instead, when this sub-block is used with the purpose of extracting the association rules, the list is kept in memory and directly used by the subsequent sub-block of the framework.

4.5.2 Association rules extraction

As mentioned before, since this sub-block is part of the same Java program of the previous one, the input is the list of critical stations produced before and already in memory. The program structure is represented in Figure 4.3, having the same characteristics of Figure 4.1 and 4.2. As in the previous section, it is recommended to look at the schema along with the following explanation to fully understand the key aspects and transformations implemented in the program.

The main steps carried out by the algorithm are:

1. The FP-Growth [11] algorithm is applied on the program input. As a reminder of Section 2.1.3, this algorithm accepts as input a list of transactions, a minimum support *minSupp* and a minimum confidence *minConf*.

   As explained before, rules extraction is a very computationally intensive task and, for some input parameters, it may need too much time or memory.
   As a consequence, when the FP-Growth algorithm is not able to converge in a limited amount of time (set to 15 minutes due to the multiple configurations that had to be executed for this thesis, but easily adjustable), the framework automatically blocks its execution, increases the minimum support by 0.1 and performs again the rules extraction. Using this approach, the minimum confidence is always set to its initial value.

   The reason behind this choice is that the rules extracted on Barcelona's dataset have very high confidence (to confirm this, see Section 5.2.3), hence the impact of the minimum confidence on the FP-Growth's execution time for Barcelona's dataset is almost negligible. Nonetheless, since for different datasets the minimum confidence may have a larger impact on the rules extraction algorithm's execution time, also this parameter is easily adjustable.

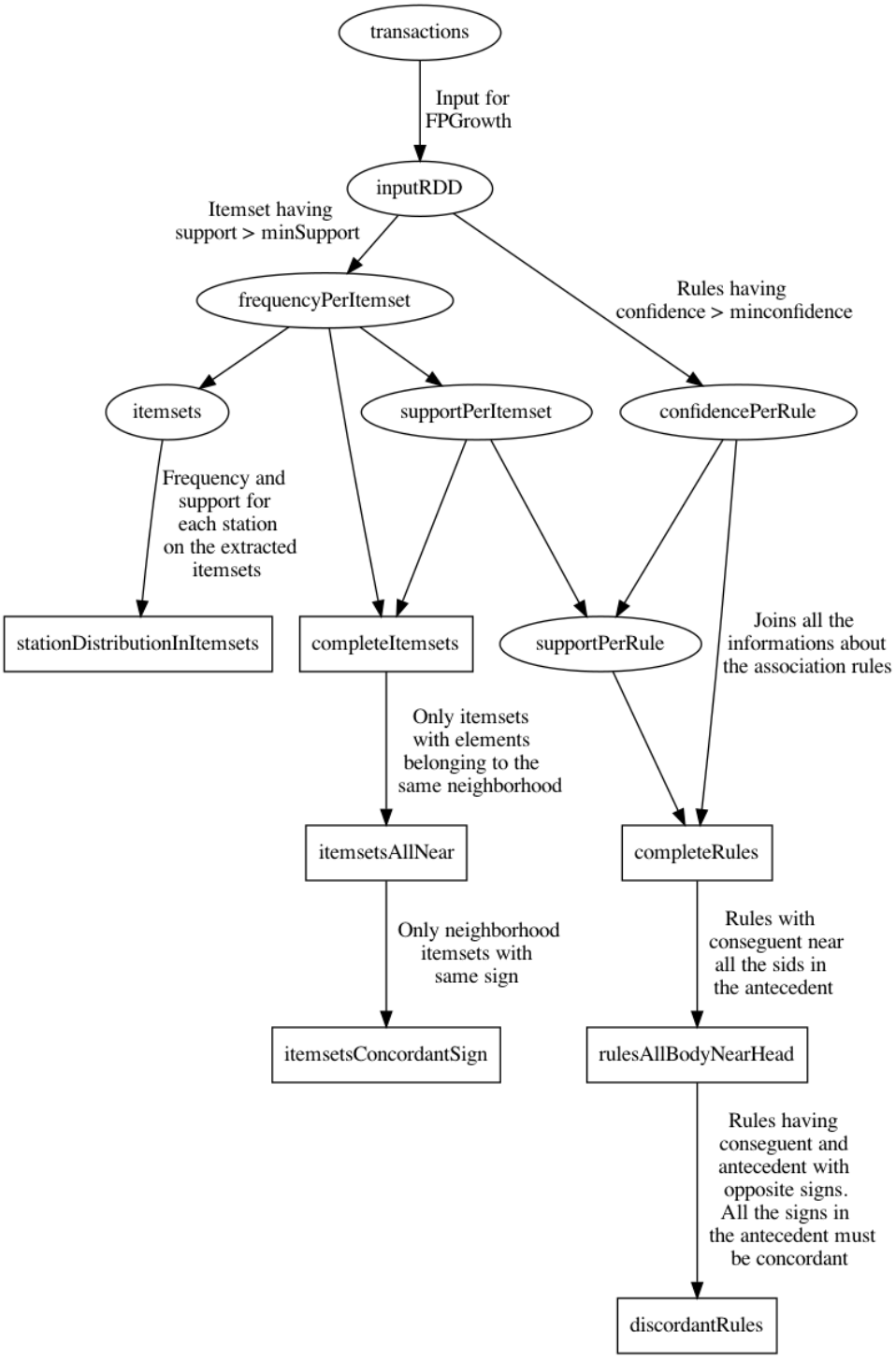

Figure 4.3. Association rules extraction JavaRDDs structure.

With reference to the program structure, the output is composed of two objects:

(a) a list of frequent itemsets along with their frequency, called *frequencyPerItemset*.

(b) a list of association rules along with their confidence, called *confidencePerRule*.

2. The support of each frequent itemset is computed and joined with the association rules, in order to generate a list of "complete" rules, i.e. rules having both support and confidence already computed and ready to be used. Furthermore, the program produces a list of "complete" itemsets, i.e. a list containing frequency and support for each frequent itemset.

   In parallel, on the left branch of the program structure represented in Figure 4.3, the itemsets are isolated from their frequencies and used to compute the percentage of itemsets containing each station. This list is called *stationDistributionInItemsets*.

3. All the complete itemsets containing only stations belonging to a common neighbourhood, are selected (*itemsetAllNear*). Then, to this subset it is applied another filter, keeping only itemsets having stations that are either all positive or negative. These itemsets are called *itemsetConcordantSign* and are one of the outputs of the program.

4. The same approach is adapted and applied to complete association rules. Since an association rule is composed of an head and a body, all the rules for which the station in the head is near to all the stations in body are selected.
   Pay attention that the MLlib [7] implementation of the FP-Growth extracts only rules having exactly one station in the consequent, so there are not edge cases in which an association rule contains more than one station in its head.

   Finally, from this subset all the rules having either a positive head and only negative elements in the body or vice versa are selected and stored as *discordantRules*. For simplicity reasons, the set of rules that respect these conditions, will be called *discordant rules*.

Looking at the program structure, it is possible to notice that this sub-block has many outputs. Each of them contain insights on the input dataset that can be exploited in different ways. In detail:

- **Discordant rules**: it is the most important information coming from this sub-block, since these rules are actively used to rebalance the target bike sharing system. How these rules are used, along with the reasons that lead to choosing this subset, is explained in Section 4.6.

- **itemsetsConcordantSign**: these itemsets are very important for bike sharing system administrators, since they represent highly problematic situations. Each itemset belonging to this subset represents a neighbourhood having a set of stations either all positive or negative, for which "trivial" rebalancing strategies such as moving bikes from positive stations to negative ones are not applicable. To fix this problem, system administrator could use this information to perform sharp changes to the system configuration, such as expanding the number of slots for some bike sharing stations.

- **stationDistributionInItemsets**: it contains the frequency and the support of each station on the extracted itemsets, where the frequency is the number of itemsets that contain s and the support is the frequency of s over the total number of itemsets.

- **completeItemsets**, **completeRules**, **rulesAllBodyNearHead**: used to compute statistics in the postprocessing phase.

- **itemsetsAllNear**: stored as output for debug purposes.

This part of the algorithm is the most computationally expensive one, being responsive for about 90% of the framework execution time. This is mainly due to the FP-Growth algorithm execution that, as stated before, is very CPU and memory intensive. Anyway, this part of the framework does not need to have bounds for its execution time, since it is executed off-line[4] on history data and just one time for a given configuration, i.e. for a given set of values for preprocessing and data analysis input parameters.

4.5.3 Postprocessing

The output produced by the Java program is then elaborated by the framework during the postprocessing phase in order to retrieve useful insights on the detected critical stations, itemsets and association rules.

The first step of this phase is to modify each file in order to fit the CSV format. In this way, python libraries such as pandas [23] can be exploited to gather insights or modify the data. On that CSV files some statistics are computed.

[4]An algorithm is said to be online if the input to the algorithm is given one at a time. This is in contrast to the off-line case where the input is known in advance [22].

In detail, the framework produces 5 CSV files:

- transactionsLength
- itemsetsLength
- rulesStatistics
- neighbourhoodRulesStatistics
- discordantRulesStatistics

The first file, *transactionsLength*, provides information about the length of the transactions generated by the *critical stations detection* sub-block. It contains the frequency of each seen transaction length, where the length of a transaction is defined as the number of critical stations in it. The second file applies the same idea to the itemsets. In fact, the file *itemsetsLength* contains the frequency of each seen itemset length, where the length of a given itemset is the number of critical stations that it contains. For example, some rows of these two files can be found respectively in Tables 4.2 and 4.3.

transaction length	frequency
40	50
41	65
42	103
43	140
44	192

Table 4.2. Few rows extracted from *transactionsLength* file.

itemset length	frequency
1	209
2	5011
3	9396
4	5262
5	1069

Table 4.3. Few rows extracted from *itemsetsLength* file.

The last 3 files are all related to association rules extracted in the previous section, one file per subset, containing information about the rules distribution per neighbourhood.
In detail, for each positively or negatively critical station, they contain the minimum, maximum and average confidence of the rules having it as head, as shown in Table 4.4. Also in this case, the distinction between positively critical and negatively critical stations is represented with the sign of the station identifier. Hence, rows having a negative station id, for example -7, contain statistics related to station 7 when it is negatively critical in association rules. Vice versa, rows with positive station identifiers identify statistics related to association rules in which the head is positively critical.

Station id	minConfidence	maxConfidence	avgConfidence
-8	0.52	0.63	0.60
-7	0.5	0.5	0.50
7	0.5	0.76	0.58
10	0.51	0.76	0.61
16	0.51	0.55	0.53

Table 4.4. Example of file structure for rules statistics.

4.6 Station rebalancing

The *station rebalancing* block is the last one of the framework, being in charge of producing the instructions for the bike sharing employees to rebalance the system. All the operations presented in this section are implemented in Python, besides the critical stations detection which is performed through the algorithm presented in Section 4.5.1.
The next subsections will justify the choices taken while developing this part of the framework and explain the steps that it performs to rebalance the system.

4.6.1 Insights analysis

Since the previous blocks gather plenty insights on the data and store them in many different output files, there were many possible information that could be exploited to perform the rebalancing. Before digging deeper in the actual rebalancing algorithm, it is important to explore the alternative ways in which the system could have been rebalanced in order to justify why it was decided to apply one of them instead of another.
In detail, there were four possible information produced by the previous block that could be exploited to solve the problem:

1. **completeRules**: the list of complete association rules could be used as it is, sorting it according to some criteria and moving the bikes among the stations included in the top N rules from the positive stations (too many bikes) to the negative ones (not enough bikes), where N is a parameter of the framework.

   This approach was discarded since there was no restriction on the position of the stations, hence rules could have contained stations very far from each other, causing two different problems:

(a) Moving bikes from two stations belonging to the same rule could have been too expensive in terms of time and fuel needed.

(b) This approach does not take into account that, since there are no difference in service offered by different bike sharing stations, usage patterns strongly depend on the position of the stations.

Any approach non position-based is affected by these two problems, so the following proposals will take it into account.

2. **ItemsetsAllNear**: these itemsets introduce a constraint for which all the stations must belong to at least one common neighbourhood. In this way, the maximum distance among two stations in the same itemset has an upper bound equal to $2 \cdot d$. As a reminder, d is a parameter of the framework introduced in Section 4.4.1 that defines the maximum size of a neighbourhood.

   This approach still does not provide an answer to a critical question: "how should the framework choose the best itemset to apply to rebalance the system?" The only quantitative measure on which it is possible to rely is the support, that, if used alone, would force the rebalancing algorithm to always prefer smaller itemsets to longer ones[5]. To solve this problem, it was decided to take advantage of another quantitative measure: association rules' confidence.

3. **rulesAllBodyNearHead**: these rules have all the properties of itemsetsAllNear, along with a new metric that can be used to have an unbiased selection criterion: the confidence.
   Anyway, the confidence is referred to the probability of a co-occurrence of body and head. As a consequence, it does not justify bike movements between elements that are contained in the body. What happens if the body of a rule contains both positively and negatively critical stations? What should be the rebalancing policy?
   Any answer to this question would have led to arbitrary assumptions, so it was decided to change approach and use only rules having either only positively or negatively critical stations in the antecedent.

4. **discordantRules**: as stated in Section 4.5.2, these rules are a subset of *rulesAllBodyNearHead* having body and head of opposite sign. Using these rules, it is possible to rebalance the system just by moving stations from head to body or vice versa, avoiding any movement inside stations contained in the antecedent. Since they are not affected by any of the problems analysed before, it was decided to exploit them to rebalance the bike sharing system.

[5]This is a direct consequence of what is called anti-monotone property, introduced in Section 2.1.

4.6.2 Rebalancing policy

Once the information to be used has been chosen, the next step is to decide how it is possible to exploit it to rebalance the system. For simplicity reasons, in the following sections "rebalancing the system through a rule r" will be shortened in "applying a rule r".

Given a rule r, the chosen rebalancing policy is to move bikes from positive stations to negative ones until the bikes are equally distributed among the stations contained in the rule. Nonetheless, since bikes are a discrete commodity, the occupancy rate of the stations in the rule may still differ after the rule application.

For instance, assume to apply a rule t 1,3 $\rightarrow -5$ in a timestamp t. During that timestamp, suppose that station 1 and 3 both have 5 taken slots and 15 free ones, while station 5 has 1 taken slot and 19 free ones. The best rebalancing for timestamp t is the one that lead each station to have an occupancy rate of $\frac{1+5+5}{20+20+20} = 0.18\overline{3}$, that corresponds to $0.18\overline{3} \cdot 20 = 3.\overline{6}$ bikes per station.

Obviously this configuration is not possible, hence the framework will use an optimal approach to minimise the occupancy rate differences among the stations. As a consequence, in the previous example, the final configuration would be: 4 bikes in station 1, 4 in station 3 and 3 in station 5.

It was decided to equally distribute the bikes because the critical definition is based on the idea of highlighting occupancy rate differences. Since the chosen rules always contain a subset of the neighbourhood of the station in rule's head, the adopted rebalancing policy tries to distance the stations in the rule from a critical condition, flattening their occupancy rate.

The framework can apply a rule r on a neighbourhood n in a given timestamp t if all the following conditions are respected:

- All stations contained in r are critical in the system at t with the same sign they have in r. It was decided to force this first condition in order to follow the usage patterns contained in the association rules. For instance, if the sign of the station was ignored, it could have been possible for the framework to remove bikes from negative stations that, at t, had more bikes than their neighbourhood, but that, according to the rule, were going to be intensively used in the near future. In this case, the rebalancing performed by the framework would be harmful, improving the configuration of the system in t, but making it unable to satisfy the user needs in the near future.
- Each station must either receive or give bikes to other stations at most one time per timestamp t. This condition was imposed in order to avoid race

conditions when rebalancing the system, hence to have a deterministic and predictable output.

- It does not exist a rule $r_2 \in R \mid \forall s_i \in r \implies s_i \in r_2$ that respects the first two conditions, where R is the set of discordant rules extracted by the previous block. The purpose of this last condition is to flatten the occupancy rate of as many stations as possible with a single rule application, since in real world systems there will always be constraints on the maximum number of rules that can be applied during a single system rebalancing. The input parameter that represent this constraint is called N and will be introduced in the following section.

If all the presented conditions hold for a rule r, then r is defined *applicable*. If instead at least one of the condition is violated, then r is defined *not applicable*.

4.6.3 Rules application

Reallocation is performed twice per day at 6:00 and 15:00, which are called *repositioning hours*. These two specific moments were chosen because typically people in Barcelona start their working day at 7:00 or later and have lunch between 14:00 and 15:00. In this way, bikes rebalancing is performed when it is more needed, so slightly before peak usages of the system. Anyway, since the whole framework was developed to be highly versatile and applicable to any bike sharing system, *repositioning hours* are received as input parameters.

The framework rebalancing process is designed to simulate the behaviour of a real world bike sharing system. To fully understand it, it is useful to analyse the framework's behavior at a specific day and repositioning hour, for instance Monday at 6:00.

The first step is loading in memory the set of rules that the framework is going to use to rebalance the system. As explained in Section 4.4.2, there are four different partitioning approaches that lead to select different bike movements. To apply all of them, the framework will execute the rebalancing algorithm four times, producing different outputs that will be compared in detail in Chapter 5.

Once the rules are in memory, they are sorted in a descendant fashion according to confidence, support and length. The first sorting parameter is confidence because rules with the highest confidence are the ones that represent the strongest patterns. In case two rules have the same confidence, it is important to exploit the pattern that occurs more frequently, hence the rule with the highest support. Finally, in case both confidence and support of two rules are the same, it was decided to

apply the longest rule because, as in the third condition for applicable rules, the framework tries to flatten the occupancy rate of as many stations as possible with a single rule application.
This sorting process orders the rules by relevance, so the framework is able to choose the best N rules for system rebalancing just by selecting the first N applicable rules.

Rebalancing planning starts at 5:00, called *filtering hour*. The framework detects all the critical stations and identifies the top N applicable rules, where N is a parameter received in input, in order to understand where the employees should be sent. As explained before, each rule represents a group of critical stations belonging to the same neighbourhood, so individuating N rules means also individuating N problematic neighbourhoods where employees are sent. Since typically one truck is enough to rebalance a single neighbourhood of stations, N is supposed to be the number of trucks available to rebalance the system. This step is performed one hour before the actual rebalancing, since it was assumed that employees of the bike sharing company would need at most one hour to move from their workplace to any bike sharing station of the city.

In this phase, it was decided to split the "month" approach in two different ones:

- **month**: in order to verify if the usage patterns of the stations were similar in the validation set, the framework aggregates the stations that are critical at *filtering hour* in any of the day of the validation week and selects among them the top N applicable rules. In simple terms, this sub-approach assumes that the configuration of the system at filtering hour is always the same, regardless of the day of the week.

- **timestamp**: in this case the framework distinguishes between the different days of the week, behaving as described previously. All the other approaches will behave in this way.

It was decided to split this approach in two in order to answer to the question "Does the usage of the bike sharing system vary according to the day of the week?". This question will be addressed in Section 5.3.5, where the performance of these two sub-approaches will be compared.

At 6:00, when rebalancing trucks are supposed to be in the target stations neighbourhood, the actual repositioning takes place. Exploiting the updated occupation data produced at 6:00[6], the framework determines which stations are critical in

[6]In a real application, these data would be retrieved from the system at 6:00. In this case, the framework simulates this behavior by gathering the status of the system at 6:00 from the validation data.

that timestamp and extracts all the rules that are still applicable from the top N ones identified at 5:00. Pay attention that, if a rule r among the N selected at 5:00 is not applicable anymore at 6:00, the framework will try to apply its largest subset in a recursive approach. If it is not possible to find an applicable rule among r and all its subsets, no rule is applied in that neighbourhood.

This restriction on applicable rules selection is forced in order to follow a behaviour reflecting as much as possible a real world application, in which employees can not move from one neighbourhood to another, maybe distant, if in the first neighbourhood there are no applicable rules.
The output is the list of movements that employees should perform, having the following structure: *x bikes should be moved from station y to station z*. All these bike movements are related to the same neighbourhood, so the employees should be able to perform the rebalancing in a short amount of time.

In a real world application, bike movements are the final output of the framework, since after their application the bike sharing system should be balanced.
Anyway, it was decided to go further and perform one last step to measure the performance of the aforementioned approaches. As a consequence, the framework simulates the rebalancing, detects again the critical stations for each repositioning hour and, in order to allow an extensive performance evaluation of each approach, outputs many different measures:

1. Total number of critical stations at repositioning hours in the validation set before rules application.
2. Total number of critical stations at repositioning hours in the validation set after rules application.
3. Number of "fixed" stations, i.e. number of stations that are not critical anymore thanks to the system rebalancing.
4. Number of bike movements needed to apply the rules.
5. Number of fixed stations per bike movement.
6. Number of times in which rebalancing the system lead to an increment in the number of critical stations, along with the timestamps in which it occurred.
7. Number of times in which the framework was not able to rebalance the system for complete lack of applicable rules.
8. Number of absent validation set slots, where a validation set "slot" is the smallest partition of the validation set that a given approach is able to select. For instance, a validation set slot for the perDay approach is "Friday", while for the perDayAndTimeSlot approach is "Friday from 5:00 to 13:00".

9. Number of slots of the validation set for which there was no data at filtering hour.

10. Number of slots of the validation set for which there was no data at repositioning hour.

It was decided to produce measures 8, 9 and 10 as *debug measures*, in order to understand the quality of the input dataset. For instance, since Barcelona's dataset is far from perfect, the system log may be incomplete, both due to problems in Barcelona's logging system and to the heavy preprocessing applied by the framework. As a consequence, typically when using the "per day and time slot" or the "per day" approach, some validation set slot may be completely empty. For instance, on September, there is no data for Monday in the validation set. This behaviour is signaled to the system administrator through measure 8. On the other hand, when the slot itself is present, it may happen that there is no available data during one of the filtering or repositioning hours, so it was decided to provide measure 9 and measure 10 in order to highlight the problem and trigger further investigations.

Chapter 5

Results

The presented framework has many different input variables that shape its behaviour in order to be flexible and to be able to rebalance any target system, hence this chapter will presents the results achieved for each block of the framework by varying all its input parameters.

The approach used to measure the effect of these parameters on the output can be divided in two parts: the first step is to choose a recommended configuration (either the best performing one or the one that best fits the bike sharing problem), so a set of default values for each of the input parameters, while the second is to vary the input parameters one at a time, in order to isolate the contribution of each of them on the output.

Finally, the performance of the different time-range based rebalancing approaches will be evaluated, in order to understand which has the best performance for Barcelona's dataset, both in terms of fixed critical stations (absolute performance) and in terms of critical stations fixed per movement (effectiveness of the approach).

5.1 Preprocessing

Since this block is in charge of cleaning the input dataset, the performance of each filter that it applies are measured in terms of deleted bike sharing stations and log lines. The idea is to find a configuration for which the cleaning is as good as possible, while keeping a representative portion of the initial dataset. As a reminder, the filters applied in this phase, along with the reasons for which they are applied, are described in detail in Section 4.4.1.

The input parameters that shape the behaviour of the preprocessing block are:

- **frequent threshold *f***: used to remove infrequent stations, i.e. stations that are not present in at least $f\%$ of the timestamps contained in the system log.
- **maximum variance *maxVariance***: used to remove stations having a number of slots that fluctuates too much, i.e. having a variance $> maxVariance$.
- **neighbourhood radius *d***: used to produce a file containing the neighbourhood of each station (*nearStations.txt*, introduced in Section 4.4.1).

While the first two parameters heavily affect the output of this block, the third one has a very limited impact, influencing only the neighbourhood's generation. Since neighbourhoods are not interesting per se but only when used to extract or to apply the association rules, both the possible values and the role of the neighbourhood radius in the framework will be discussed in the next blocks.

5.1.1 Preprocessing reference configuration

In order to measure the effects of each input parameter on the preprocessing phase, it is necessary to choose a configuration that will act as a reference when varying the values of the input parameters. As it will be shown below, the values that guarantee the best compromise between quality of the data and number of filtered rows are 90% for the frequent threshold f and 5 for the maximum variance *maxVariance*. Table 5.1 shows some statistics computed on the output of this configuration.

Metric	Value
Initial number of stations	3301
# stations not present in any timestamp	3017
# not frequent(infrequent) stations	8
# unstable stations	155
# accepted stations	121
Initial number of rows	25319028
# rows with 0 free and 0 used slots	214907
# rows with 1 total slot	67847
# rows related to infrequent stations	473965
# rows related to unstable stations	13760988
# rows with incomplete timestamps	1810416
# clean log rows after preprocessing	8990905

Table 5.1. Preprocessing results obtained with frequent threshold $f = 90\%$ and maximum variance $maxVariance = 5$. These two parameters affect only the highlighted rows, so these metrics will be the only ones used to compare the performance of the preprocessing block for different values of the input parameters.

Furthermore, in order to ease the data interpretation, it was decided to produce two additional pie charts, respectively showing the stations related metrics (Figure 5.1) and the log related metrics (Figure 5.2).

It is then possible to notice that the reference configuration filters out a very high number of stations (3180, 96.3% of the total), but this is mainly due to the fact that the large majority of the initial stations (3017, 91.4% of the total) is not able to log its status. On the other hand, the percentage of filtered log lines is much lower (65%), so it is reasonable to say that the remaining ones (35%, about $9 \cdot 10^6$ log lines) are enough to be still representative of the behaviour of Barcelona's bike sharing system.

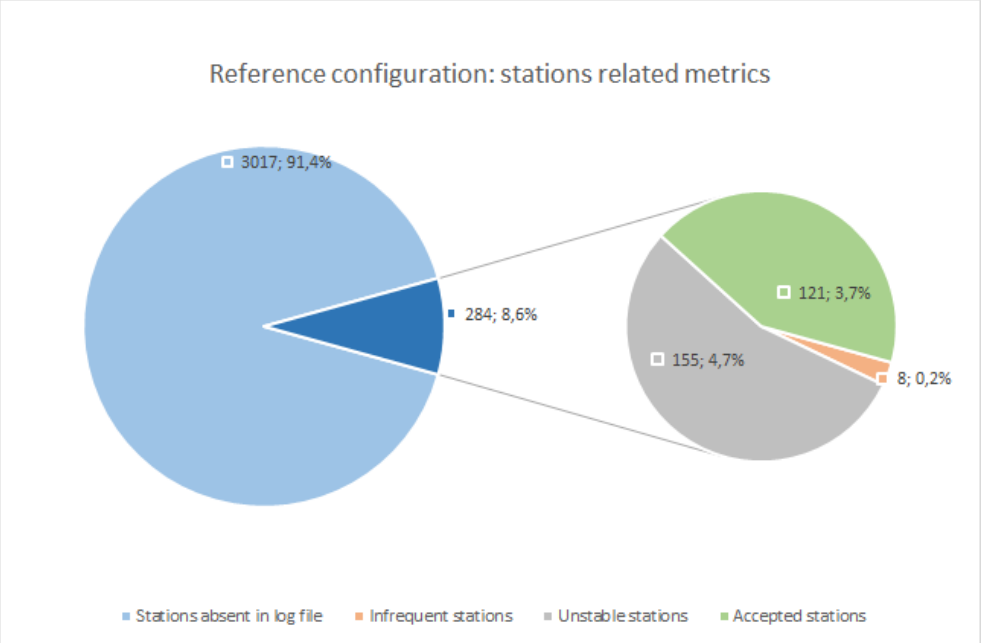

Figure 5.1. Stations metrics for the reference configuration.

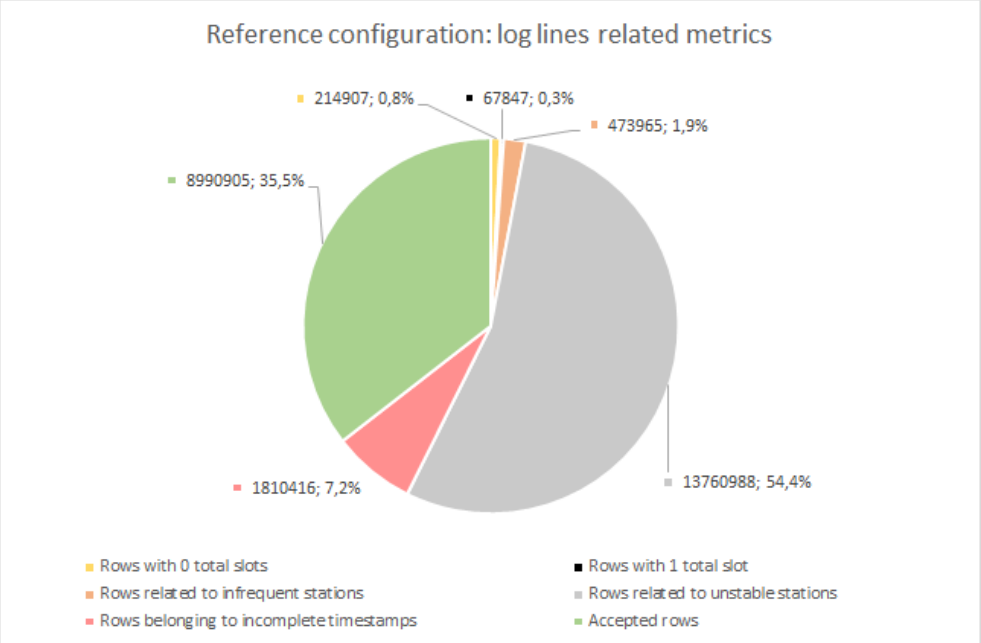

Figure 5.2. Log metrics for the reference configuration.

5.1.2 Effect of frequent threshold *f*

Having chosen a default configuration, it is possible to evaluate the effect of the frequent threshold f on the preprocessing output by setting $maxVariance = 5$ and varying the value of f. Since f represents the percentage of timestamps in which a station must log its status in order to be kept, it was decided to try the following values for f: 10%[1], 80%, 90% and 95%. For these values, the preprocessing output is described by the values at Table 5.2.

Metric	f=10%	f=80%	f=90%	f=95%
# infrequent stations	1	4	8	12
# unstable stations	162	159	155	151
# accepted stations	121	121	121	121
# rows related to infrequent stations	165	155988	473965	804105
# rows related to unstable stations	14234788	14078965	13760988	13430848
# rows with incomplete timestamps	1810416	1810416	1810416	1810416
# clean log rows after preprocessing	8990905	8990905	8990905	8990905

Table 5.2. Preprocessing results obtained with variable frequency threshold and maximum variance $maxVariance = 5$. Pay attention that the white rows present in Table 5.1 are not present here, because their values do not depend on the frequency threshold.

It is possible to notice that, even if the value of the frequency threshold f varies, the number of clean log rows and accepted stations stays the same. The reason is that, in Barcelona's dataset, *infrequent* stations are also *unstable*. Hence, if these stations are not filtered by the frequency threshold, they are removed from the *maxVariance* filter. To confirm this, it is possible to look at the number of unstable stations, for which holds $\#infrequent\ stations + \#unstable\ stations = costant$. As a consequence, it was decided to keep the frequency threshold to 90%, a reasonable value for a bike sharing system. Using this value, infrequent stations like station 24[2], which is present in about 73% of the timestamps, are filtered out.

Pay attention that these considerations on the frequency threshold are correct in this very specific case, but in other systems they could be wrong! Subsequently, it was decided to keep the frequency threshold as an input parameter, to be able to tailor the framework to any type of target system.

[1] Mainly for debug purposes, introduced to understand if the filter was working properly.

[2] The number of slots of station 24 is represented at Figure 3.2 at page 20. It is easily possible to notice that the number of timestamps of that station is much lower than the ones of station 8 or even 29, respectively represented in Figure 3.1 at page 19 and Figure 3.3 at page 20.

5.1.3 Effect of variance threshold *maxVariance*

Moving on to the *maxVariance* parameter, it was decided to try three different values: 3, 5 and 7. These three values were chosen by looking at the *meanAndVariance.txt* file presented in Section 4.4.1, which contained mean and variance for each *frequent* bike sharing station in Barcelona. Since the filter on the variance of the stations is applied after the one on the frequency of the stations, the value of maxVariance does not affect the number of stations and log rows removed by the latter. Hence, the results displayed in Table 5.3 do not include these metrics. Anyway, for a complete comparison, it is sufficient to fill the missing rows with the values present in Table 5.1.

Metric	**maxV=3**	**maxV=5**	**maxV=7**
# unstable stations	232	155	114
# accepted stations	44	121	162
# rows related to unstable stations	20628837	13760988	10115164
# rows with incomplete timestamps	220620	1810416	5694123
# clean log rows after preprocessing	3712852	8990905	8753022

Table 5.3. Preprocessing results obtained with frequency threshold $f = 90\%$ and variable maximum variance.

Looking at Table 5.3, it is immediately noticeable that the number of log rows produced by the preprocessing phase vary according to the value of maxVariance. Therefore, this filter has an impact on the performance of the preprocessing phase when the framework is applied to Barcelona's dataset.

When $maxVariance = 3$, the variance threshold alone filters out more than 81% of the initial $2.5 \cdot 10^7$ log lines, making the preprocessing phase produce only about $3.7 \cdot 10^6$ clean log lines. While the quality of the output data is certainly very good, the number of remaining rows is too low to be representative of the initial dataset.

On the other hand, setting $maxVariance$ to 7 makes the variance threshold filter way less rows ($\approx 10^7$) than the ones removed for $maxVariance = 3$ ($\approx 2.1 \cdot 10^7$) or $maxVariance = 5$ ($\approx 1.4 \cdot 10^7$), but the remaining stations are sparse in the available timestamps. As a consequence, the number of timestamps not containing all the *stable* and *frequent* stations is much higher than for the other configurations, resulting in a number of clean log lines (8753022) that is inferior to the ones produced using $maxVariance = 5$ (8990905), while requiring a lower level of accuracy in the data.

In conclusion, since in this specific case there is no disadvantage in keeping a more strict variance filter, it was decided to set $maxVariance = 5$ for the maximum variance threshold, while the frequency threshold is set to 90%, even if its value does not affect the performance of the framework for Barcelona's dataset.

The effect of the chosen configuration on station 8, the only one that the preprocessing keeps among the three represented in Section 4.4.1, is represented in Figures 5.3 and 5.4, which respectively display the number of slots of that station before and after the preprocessing.

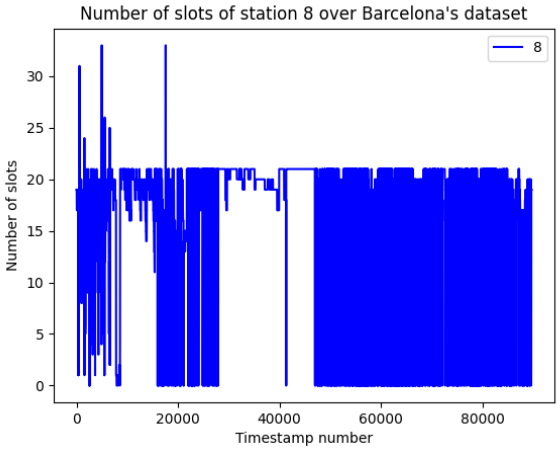

Figure 5.3. Number of slots of station 8 before the preprocessing.

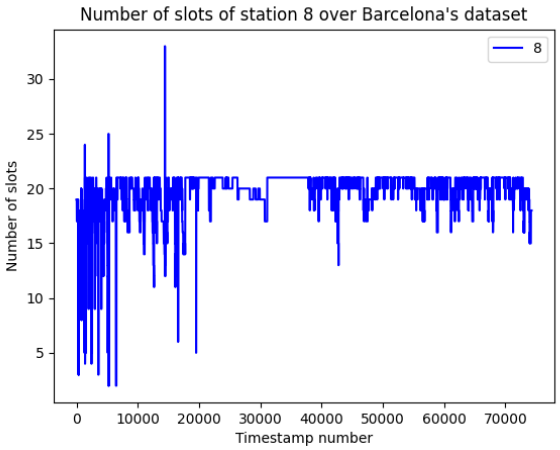

Figure 5.4. Number of slots of station 8 after preprocessing.

5.2 Rules Extraction

Once the preprocessing is done, the framework enters in the rules extraction phase. While the input parameters of the preprocessing had to be tailored in order to perform an accurate data cleaning while keeping the output representative of the dataset, this part of the framework does not have any type of constraints. Indeed, the purpose of this block is to extract a reasonable number of *meaningful* rules, but while their number is immediately available, their quality is measurable only looking at the performance of the station rebalancing block. Nonetheless, it is still possible to make some considerations on how the rules extraction input parameters affect length and number both of detected critical stations and of extracted rules. This block has four different input variables:

- **neighbourhood radius *d***: determines the size of a neighbourhood, hence it affects both the detected critical stations and the rules extracted from them.

- **critical threshold *t***: it defines when a station s is critical or not, according to the difference between its occupancy rate and the average occupancy rate of its neighbourhood. Therefore, this parameter influences both the detected critical stations and the rules extracted from them.

- **minimum support *minSupp*** and **minimum confidence *minConf***: they are parameters of the FP-Growth algorithm, so they directly affects the association rules extracted.

Pay attention that, while the minimum support and confidence are only related to the framework and can be modified at will to obtain better performance in terms of system rebalancing, the neighbourhood radius and the critical threshold define the rebalancing problem itself. In fact, these two parameters affect the core concepts presented in Section 4.2, such as the meaning of *critical* station and *neighbourhood* for the framework itself.

As a consequence, when these two parameters will be set to different values in order to measure their effects on the output, the results will determine if the problem represented by these values is more or less complex than the reference one, more than if that configuration is more or less effective than the reference one.

5.2.1 Rules extraction reference configuration

Since the quality of the rules extracted in this block can be measured only observing the final output of the framework, the best configuration will be the one that on one hand best models the rebalancing problem for Barcelona's bike sharing system and on the other is able to solve that same problem with the highest performance. In detail:

- **Neighbourhood radius**: defining the dimension of a bike neighbourhood, it has an high impact on the usability of the system for end users: the larger the neighbourhood, the larger the *almost critical* acceptable configurations, i.e. a configuration in which all the bikes are positioned at one extremity of the neighbourhood. Since each station has its own neighbourhood, if the framework consider a given neighbourhood to be balanced, the maximum distance that the end user has to travel from an empty(full) station to rent(park) a bike is equal to the neighbourhood radius. Hence, it was decided to set the neighbourhood radius value to **500 metres**, which seems a reasonable distance to travel in order to rent or park a bike.

- **Critical threshold**: defines when a station is critical, so it must be set to a value able to highlight only non negligible discrepancy between the occupancy rate of a station and the average one of its neighbourhood. On the other hand, a critical station must be detected before its occupancy rate becomes too low(high) with respect to the average of its neighbourhood, becoming useless for end users. Therefore, it was decided to set the reference value of the critical threshold to **20%**, which seems a reasonable trade-off. For instance, in a neighbourhood having, on average, 30 bike slots per station, a station should have at least 6 bikes more(less) of the average in order to be positively(negatively) critical.

- **Minimum support and minimum confidence**: these two values are discussed together since they affect the same part of the framework: the FP-Growth algorithm. The values of these two parameters are respectively 0.1 and 0.5, the lowest that lead the FP-Growth algorithm to converge for at least some of the months and approaches used.
  It was decided to use the lowest possible values for these two parameters in order to show their effect on the rules extracted using the largest set of association rules that the algorithm is able to extract from Barcelona's dataset, but is not assumed that these two values will be the best ones to rebalance the bike sharing system. Hence, both the minimum support and the minimum confidence will be analysed again (and possibly modified) to find the best performing framework configuration.

5.2.2 Effect of neighbourhood radius and critical threshold

As stated before, neighbourhood radius and critical threshold are parameters of the rebalancing problem more than of the framework itself. As a consequence, setting them to different values models different real-world problems, instead of affecting the framework's performance. Given that the problem the framework has to solve is to rebalance Barcelona's bike sharing system, it was decided to consider the following possible values for neighbourhood radius and critical threshold:

- **neighbourhood radius**: 500 metres and 1km, because higher values would have forced the end user to walk too much just to rent or park a bike in *almost critical* configurations, while lower values would have left some bike sharing stations alone in their neighbourhoods.
- **critical threshold**: 10, 20 and 30 percent, because higher or lower values would have led the framework to solve a problem that does not model well Barcelona's bike sharing system rebalancing problem. Assuming to operate on a neighbourhood composed by stations having 30 bike slots on average, a discrepancy of 5% in the occupancy rate is negligible, representing a discrepancy of 2 bikes (1.5 to be accurate). On the other hand, assuming to operate in that same neighbourhood, a discrepancy of 50% represents a discrepancy of 15 bikes with respect to the average, hence it should have already been addressed.

These two parameters affect the concepts of *critical* station and *neighbourhood*, and, consequently, the rules extracted by the framework. Therefore, the impact of changes in either the neighbourhood radius or the critical threshold are measurable through two metrics: number of critical stations detected per timestamp and number of rules extracted by the framework for a given partition of a given approach[3].

Therefore, it was decided to represent the first metric on two distinct charts, one per parameter, with the number of critical stations on the x axis and the number of timestamps having that number of critical stations on the y axis. The former, represented in Figure 5.5, contains two functions, drawn setting the critical threshold to the reference value (20%) and varying the neighbourhood radius among 500 metres and 1km. The latter, represented in Figure 5.6, contains three functions, obtained setting the neighbourhood radius to the reference value (500 metres) and varying the critical threshold among its three possible values: 10%, 20% and 30%.

[3]For instance the rules extracted by the perDay approach, month August, day Friday.

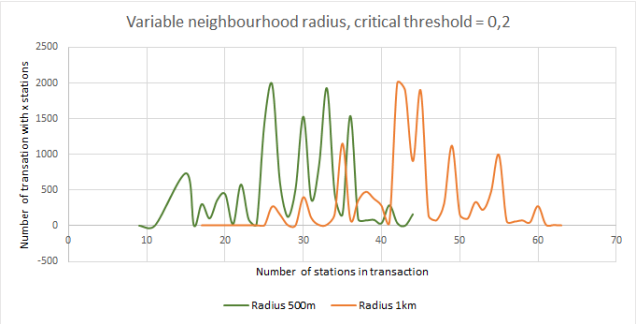

Figure 5.5. It is possible to notice that the two curves are very similar, but the one having neighbourhood radius = 1km is shifted to the right, having, on average, an higher number of critical stations per timestamp with respect to the other curve.

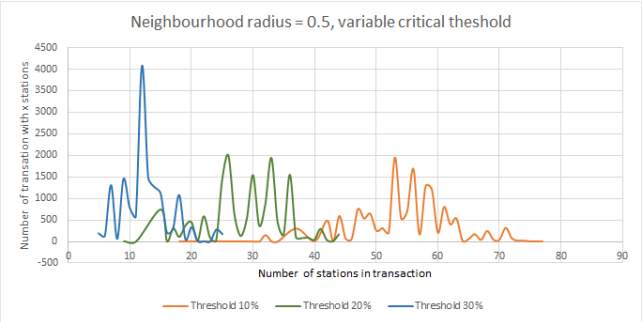

Figure 5.6. It is possible to observe that lower values of the critical threshold shift the curves to the right, due to the fact that lowering the threshold that establish when a station is critical, the number of critical stations per timestamp can only grow, i.e. stations that are critical with a 20% threshold must be critical also with a 10% threshold.

From these two figures is possible to observe that high values of the neighbourhood radius and low values of the critical threshold cause an increase in the average number of critical stations per timestamp. This behaviour was largely expected, since an increase in the neighbourhood radius produce neighbourhoods more intersected with each other, hence an higher number of criticality checks for each station. For instance, if with a neighbourhood radius equal to 500 metres station 1 is in the neighbourhoods of station 3 and 4, with a neighbourhood radius of 1km station 1 may also be contained in the neighbourhoods of station 6 and 7, being more frequently critical. Moreover, a decrease in the critical threshold cause the minimum accepted variation in the occupancy rate to shrink as well, leading the framework to detect more critical stations per timestamp.

Furthermore, to measure the impact of the neighbourhood radius and of the critical threshold on the rules extracted by the framework, it was decided to produce two additional charts (again, one per parameter) representing the number of discordant rules extracted by the framework while varying either the neighbourhood radius or the critical threshold.

The former, represented in Figure 5.7, contains the number of discordant rules extracted by the framework for neighbourhood radius equal to 500 metres and 1km, having set the critical threshold to its reference value (20%).

The latter, displayed in Figure 5.8, represents the same measure for three possible values of the critical threshold (10, 20 and 30%), having set the neighbourhood radius to its reference value (500 metres).

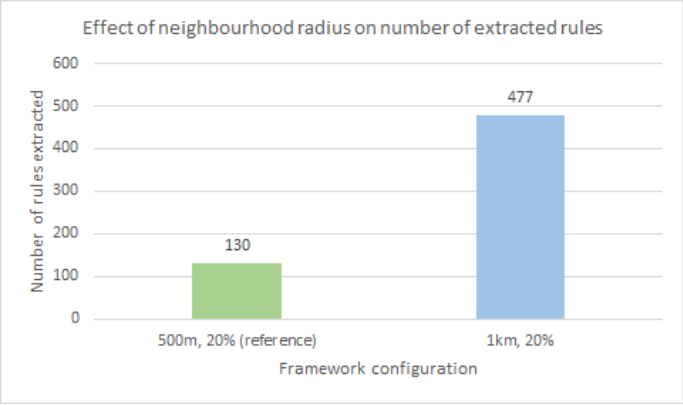

Figure 5.7. Number of rules extracted by the framework for Mondays in June (perDay approach) setting the neighbourhood radius to 500 or 1km.

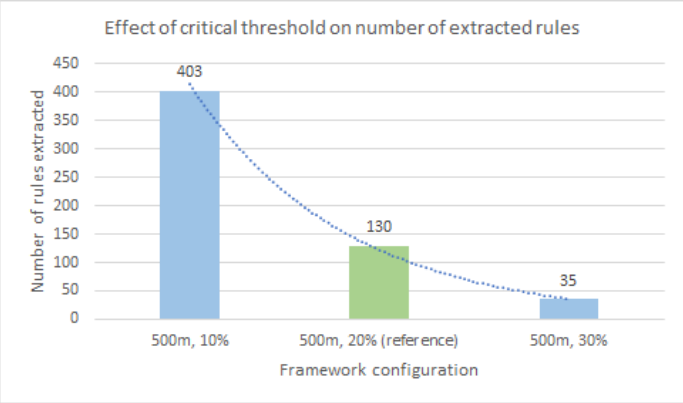

Figure 5.8. Number of rules extracted by the framework for Mondays in June (perDay approach) setting the critical threshold to 10%, 20% or 30%.

Both these charts are referred to the discordant rules extracted by the **per day** approach from the data related to **Mondays** in **June**. These specific month, day and approach were chosen both because the FP-Growth algorithm converged with a minimum support of 0.1[4] and because the number of rules extracted was 130, high enough to be significant.

These two charts confirm what observed in Figures 5.5 and 5.6. In detail, since a larger neighbourhood radius produces a larger set of critical stations per timestamp, also the number of rules extracted by the algorithm grows. Furthermore, a smaller critical threshold leads to more critical stations per timestamp, therefore to an higher number of rules. It is also interesting to notice that the number of extracted rules seem to follow an exponential trend in function of the critical threshold.

5.2.3 Effect of minimum support and minimum confidence

The remaining input parameters of the rules extraction block are the minimum support and the minimum confidence thresholds. Also in this case, the real effect of these two parameters can be measured only looking at the performance of the whole framework, since the association rules in this block become useful only when they are applied to rebalance the system. Nonetheless, it is possible to analyse the effect of these two parameters on the number of rules extracted.

In order to do that, it was decided to represent the number of extracted rules varying the minimum support (Figure 5.9) and the minimum confidence (Figure 5.10). These two charts represent the value of the minimum support(minimum confidence) on the x axis, while the y axis represents the percentage of extracted rules having a support(confidence) higher or equal to the x axis value. In order to be coherent with the charts generated for the neighbourhood radius and the critical threshold, also in this case the figures are referred to the rules extracted by the *per day* approach to rebalance Mondays in June.

These two charts confirm that minimum support and minimum confidence influence in a very different way the number of rules extracted by the framework.

It is possible to notice that the curve representing the percentage of rules extracted for different values of the minimum support (Figure 5.9) follows an exponential trend, keeping only 30% of the extracted rules just by increasing the support value from 0.1 to 0.2. Furthermore, only 10% of the rules have a support higher than 0.25.

[4]It is not obvious, see Section 4.5.2.

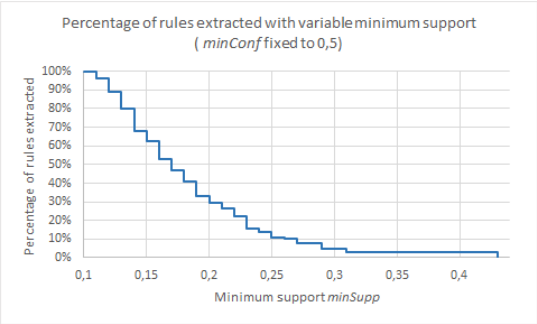

Figure 5.9. Percentage of rules extracted for variable minimum support by the *per day* approach on June Mondays.

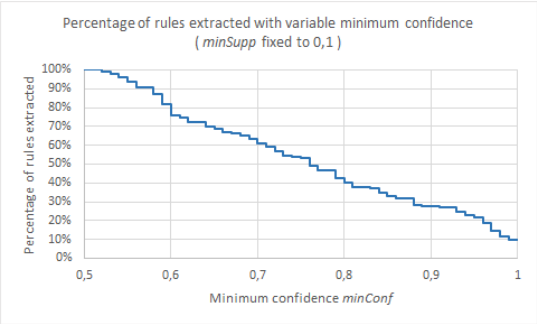

Figure 5.10. Percentage of rules extracted for variable minimum confidence by the *per day* approach on June Mondays.

On the other hand, It is possible to observe that the curve representing the percentage of rules extracted for different values of the minimum confidence (Figure 5.10) is much less sharp than the one generated by varying the minimum support, resembling more a linear function than an exponential one. In fact, while modifying the minimum support from 0.1 to 0.2 filters out 70% of the rules, an increase of the minimum confidence from 0.5 to 0.6 removes less than the 20% of the rules.

The different trends observed in the percentage of extracted rules when varying either support or confidence justify the decision of varying only the minimum support in case the FP-Growth algorithm is not able to converge in a limited time, since, for Barcelona's dataset, changing the value of the minimum confidence would not have helped the FP-Growth algorithm to converge[5].

[5]To be exact, *small* changes in the minimum confidence would not have helped. Obviously, setting the minimum confidence to 0.95 would have helped the FP-Growth algorithm to converge, but it would also have led to extracting a very limited number of rules representing hyper specific (hence useless) patterns.

5.3 Station rebalancing

The last step is then to actually rebalance Barcelona's bike sharing system through the discordant association rules extracted in the *rules extraction* block.

Since the output of this block is also the final output of the framework, it will depend on all the framework's input parameters. As a reminder, this is a complete list of the framework's parameters:

- **frequent threshold *f***: detects *infrequent* stations in preprocessing, set to 0.9.
- **variance threshold *maxVariance***: detects *unstable* stations in preprocessing, set to 5.
- **neighbourhood radius *d***: used during the preprocessing block, but analysed in the rules extraction block (Section 5.2). Since it affects the rebalancing problem itself, it is important to analyse its effect on the system rebalancing proposed by the framework. Its value will be either 500 metres or 1km, for the reasons exposed in Section 5.2.2.
- **critical threshold *t***: used to detect the critical stations (sub-block of rules extraction, Section 4.5.1), also this parameter describes the rebalancing problem itself, hence it makes sense to observe how it affects the final system rebalancing. It will be set to 10, 20 or 30%, as discussed in Section 5.2.2.
- **minimum support *minSupp***: it is used in the rules extraction block (input of the FP-Growth algorithm), but since these rules are used to rebalance the bike sharing system, it affects also the final output. The previous section did not propose any discrete value for this parameter, since its effect on the output of the rules extraction block was analysed in terms of a continuous function describing the percentage of rules extracted by the FP-Growth algorithm for a minimum support $minSupp \mid 0.1 \leq minSupp \leq 1$ (Figure 5.9).
  Obviously, this parameter must be set to a discrete value in order to be used in the framework, so its possible values will be discussed below.
- **minimum confidence *minConf***: as *minSupp*, it is an input parameter of the FP-Growth algorithm and was analysed in the same way in Section 5.2.3. Its effects on the output, along with the values to which it can be set, will be discussed below.
- **repositioning hours**: this parameter represents the moments of the day in which the system rebalancing takes place. As exposed in Section 4.6.3, the *repositiong hours* were selected to fit Barcelona's citizens lifestyle, hence, to

rebalance this specific bike sharing system, this parameter will be considered constant [6].

- **number of trucks N**: it is the number of trucks that the bike sharing system will use to rebalance the system. This parameter is used only in the last block of the framework, so its possible values will be discussed below.

Among all input parameters, the ones that still lack a set of input values are the minimum support, the minimum confidence and N.

For what concerns the minimum support: given that 0.1 was the lowest value that enabled the FP-Growth algorithm to converge and since it was observed that small variations in the minimum support hugely affect the number of rules extracted, it was decided to set *minSupp* to 3 values: 0.1, 0.2 and 0.3. This set of values allows the FP-Growth algorithm to converge, while still extracting an high number of association rules.

Moving to the minimum confidence: given that the framework uses the applicable rules with the highest confidence to rebalance the system and since for Barcelona's dataset the effect of the minimum confidence on the number of extracted rules is very limited, it was decided to set *minConf* to the lowest value that let the FP-Growth algorithm converge, that is 0.5. Higher values would not have given a real contribution to the output, filtering out rules that the framework did not already use, while lower values would have been too much of a burden for the FP-Growth algorithm, making its execution too computationally expensive.

Finally, since the parameter N represents the number of trucks used to rebalance the bike sharing system, it is reasonable to assume that this parameter could be equal to 5, 10 or 20 for a large city such as Barcelona.

Given these considerations, the parameters that will be considered **variable** in this section are:

- **neighbourhood radius d**: 500 metres or 1km.
- **critical threshold t**: 10%, 20% or 30%.
- **minimum support *minSupp***: 0.1, 0.2 or 0.3.
- **number of trucks N**: 5, 10 or 20 trucks.

[6] Also the time interval between *filtering* and *repositioning hour* is a parameter, but in this case it will be always set to 1 hour, which gives to the bike sharing system's employees a reasonable time in order to move to the target neighbourhoods.

5.3.1 Best performing configuration and approach

Since this block is in charge of rebalancing the system, the best performing configuration and approach will be chosen looking both at the number of fixed bike sharing stations, which represents the absolute performance of the framework, and at the number of fixed stations per movement, which represent the effectiveness of the corrective actions proposed by the framework. The chosen configuration will then be used as a reference, to show the effects of each of the 4 aforementioned variable parameters.

As explained multiple times in the previous chapters, the neighbourhood radius and the critical threshold describe the **rebalancing problem** itself, hence they are not considered when choosing the best performing configuration. For this reason these two parameters are respectively set to two specific values: respectively 500 metres and 20%[7]. Nonetheless, it was decided to execute the rebalancing for all the 54 possible configurations[8] to check the performance of the framework when solving different problems.

From a detailed analysis of the data produced by the framework, the best performing configuration and approach for Barcelona's bike sharing system is the **perDay** one when used with the following input parameters:

- **neighbourhood radius *d***: 500 metres.
- **critical threshold *t***: 20%.
- **minimum support *minSupp***: 0.1.
- **number of trucks *N***: 5.

As displayed in Figure 5.11, using this configuration the **perDay** approach is able to fix 225 stations over a total of 1377 critical ones by performing only 151 bike movements, which means that, on average, each bike movement fixes 1.49 stations (displayed in Figure 5.12). The perDay approach is able to achieve these performance despite being forced to skip the system rebalancing for 8 times on a total of 43[9], hence it is reasonable to say that with a larger input dataset it could be

[7] A detailed explanation of this choice can be found in Section 5.2.1.

[8] There are 2 values for neighbourhood radius, 3 values for critical threshold, 3 values for minimum support and 3 values for number of trucks: $2 \cdot 3 \cdot 3 \cdot 3 = 54$ possible configurations.

[9] The input dataset should have allowed the framework to rebalance Barcelona's bike sharing system a total of 70 times: 2 repositioning hours · 7 days in the validation set · 5 months. Nevertheless, many timestamps were missing in the dataset, hence it was possible to rebalance the system just 43 times.

able to perform even better. Furthermore, this configuration and approach never lead to a growth in the number of critical stations when rebalancing the system (output number 6 of the stations rebalancing block).

As it will be shown in the next sections, this specific configuration and approach were chosen because they were the best compromise between performance and efficiency to solve this specific rebalancing problem, characterised by a neighbourhood radius of 500 metres and a critical threshold of 20%. Moreover, the perDay approach will prove to be the best performing one in terms of number of stations fixed in all the tried input configurations, while being always in the top two best performing approaches in terms of stations fixed per movement.

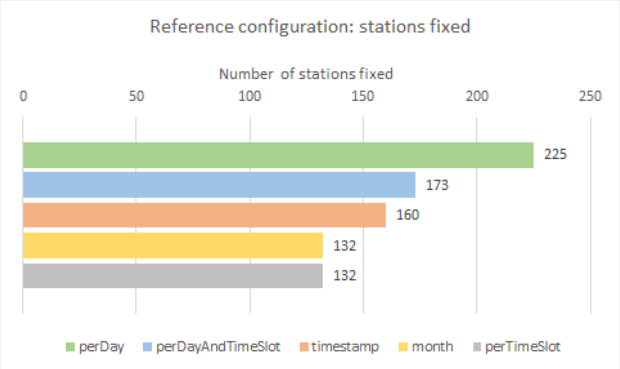

Figure 5.11. Stations fixed by each of the 5 approaches using the best performing configuration. It is possible to notice that the perDay approach fixes 20% more stations than the second best performing approach (perDayAndTimeSlot).

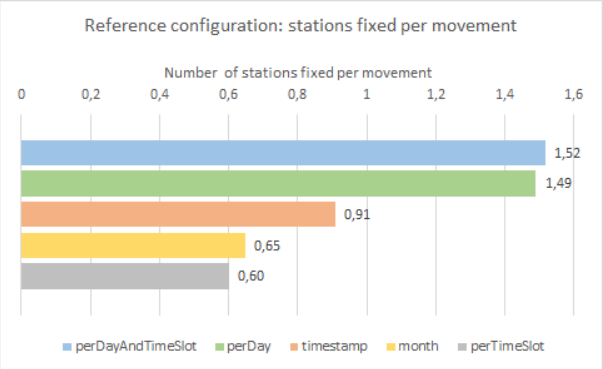

Figure 5.12. Stations fixed per movement by each of the 5 approaches using the best performing configuration. In this case, the approach perDayAndTimeSlot is slightly more efficient ($\approx +2\%$) than the perDay one, but the two values are totally comparable.

5.3.2 Effect of neighbourhood radius *d*

As stated before, the neighbourhood radius defines the rebalancing problem itself. Hence, to understand if for a given value of the neighbourhood radius the problem has become more difficult or easier to solve, it would not make sense comparing the number of stations fixed by two different configurations, since they have completely different meanings. For instance, assume that a given configuration fixes 300 critical station, while the best configuration is able to fix 225. Looking only at these numbers, the new problem seem to be easier to solve, but what if the new configuration detects 3000 critical stations instead of the 1377 critical stations detected by the reference configuration? Hence, in order to be able to compare the difficulty of different problems, it was decided to introduce a new metric that could link these two measures: the **percentage of stations fixed** by the framework. This approach will be used both for the neighbourhood radius and for the critical threshold, because, since they both model the problem that the framework has to solve, any variation in these two parameters will make the framework detect a different number of critical stations.

As explained in Section 5.2.2, the neighbourhood radius can take two possible values: 500 metres and 1km. Hence, there is just one alternative configuration that will be compared to the reference one, having the default values for all the input parameters except for the neighbourhood radius, that is set to 1km. The percentage of stations fixed by these two configurations is shown in Figure 5.13.

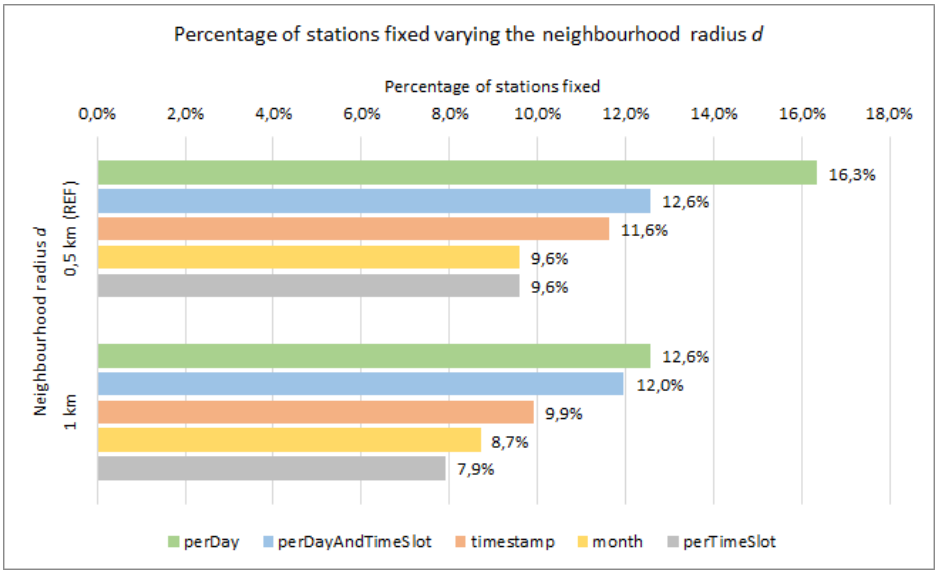

Figure 5.13. Percentage of stations fixed for different values of the neighbourhood radius.

It is then possible to observe that the percentage of stations fixed by all the approaches decrease when the neighbourhood radius increases from 500 metres to 1km. Therefore, it is reasonable to say that the alternative configuration represents a more complex problem than the one outlined by the reference one. This behaviour was expected, because increasing the neighbourhood radius implies an higher number of criticality checks for each station, hence an higher probability that changing the occupancy rate of a station generates other critical stations in adjacent neighbourhoods.

5.3.3 Effect of critical threshold *t*

The critical threshold is the second (and last) parameter that affects the problem definition, hence all the considerations performed for the neighbourhood radius still hold. It is then interesting to compare the percentage of stations fixed by the framework in the reference configuration to two alternative configurations, obtained setting the value of the critical threshold respectively to 10% and 30%. The performance of the system rebalancing for these three configuration are then represented in Figure 5.14.

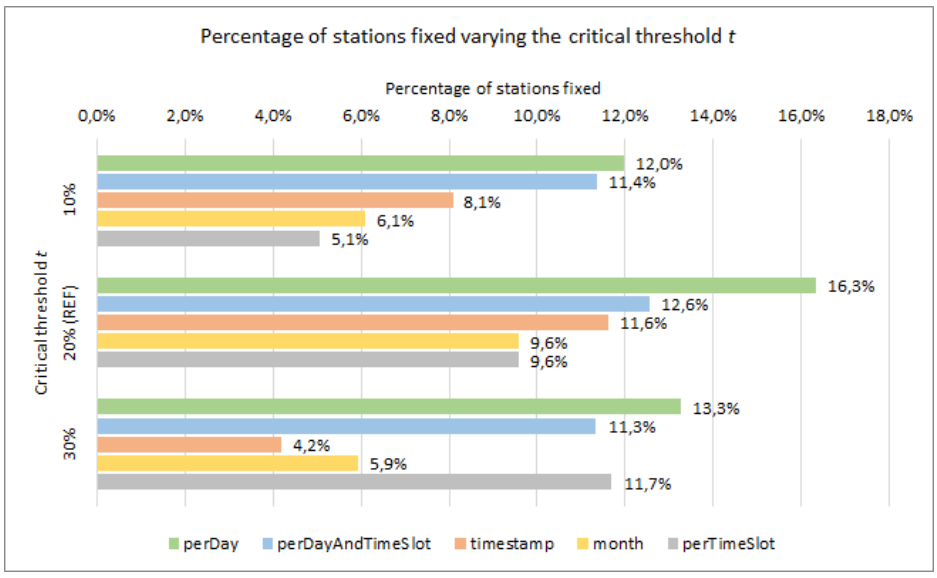

Figure 5.14. Percentage of stations fixed for different values of the critical threshold.

Looking at this chart it is then possible to observe that the rebalancing problem becomes more difficult to solve for values of the critical threshold different from the reference one (20%). In this case, even looking more in depth at the data produced by the framework, it was not possible to find simple qualitative explanation for this

behaviour, since modifying the critical threshold has both positive and negative effects on the difficulty of the problem.

Furthermore, it seems that the perTimeSlot approach does not follow the trend of the other approaches, having higher and higher performance for increasing values of the critical threshold. Nevertheless, this approach is not able to outperform the perDay approach for any value of the critical threshold, being able to fix a lower number of critical stations than the reference approach in each of the three analysed configurations.

5.3.4 Effect of minimum support *minSupp*

From this point on, the neighbourhood radius and the critical threshold will be set to their default value (respectively 500 metres and 20%), hence the performance of the framework achieved by the configurations analysed in this and in the next section will be compared through the total number of stations fixed (absolute performance) and through the number of stations fixed per movement (relative performance, efficiency of the framework). It was decided to use a different approach with respect to the one used for the previous two parameters because, having set the neighbourhood radius and the critical threshold to the same values for all the configurations, the framework is going to solve the **same problem** independently from the value of the minimum support and of the number of trucks. As a consequence, the initial number of critical stations will be constant across all the configurations, therefore using the percentage of fixed stations will not be needed anymore.

In this section, the two alternative configurations will have the same input parameters of the reference one, besides their minimum support value, either set to 0.2 or 0.3, which in the reference configuration is set to 0.1.

As it is possible to observe from Figure 5.15, in both the alternative configurations an increase in the minimum support causes the total number of fixed stations to drop. Nevertheless, the best performing approach remains the **perDay** one, as evidence of the effectiveness of the rules extracted by this approach.

This trend was totally expected, since an increase in the minimum support causes the number of extracted rules to sharply decrease, hence the number of neighbourhoods rebalanced by the framework decrease as well.
Nonetheless, it is possible to notice that the number of stations fixed by the perTimeSlot and perDayAndTimeSlot approaches do not degrade as fast as the other approaches. Looking more in depth at the data, for what concerns the perDayAndTimeSlot approach, the missing decrease in the number of fixed stations while

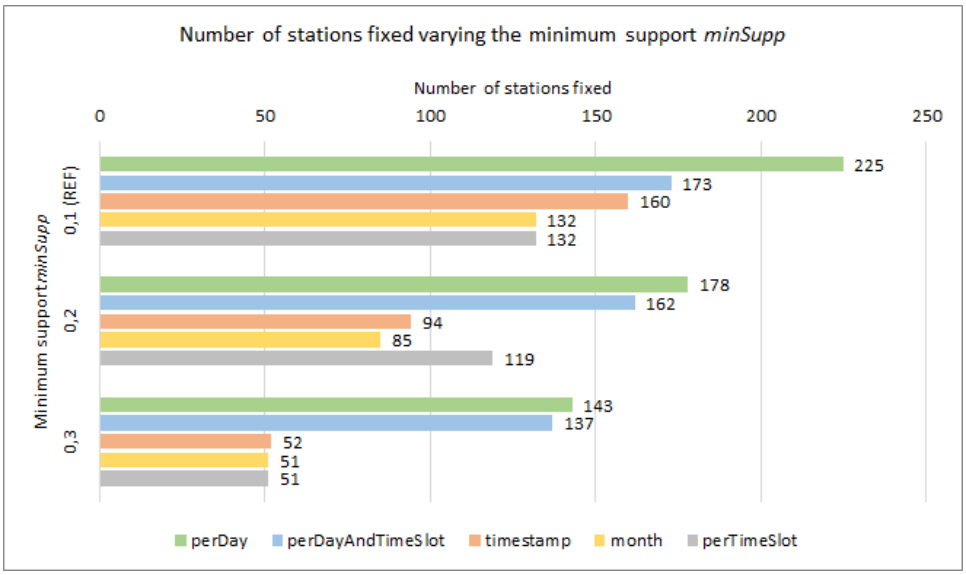

Figure 5.15. Number of stations fixed for different minimum supports.

setting the minimum support to 0.2 is due to a much smaller decrease in the number of applied rules (72 instead of 91, -21%) with respect to the other approaches (month/timestamp -36%, perDay -30%, perTimeSlot -28%). Anyway, this consideration alone is not enough to justify that much of a difference. As a consequence, the second consideration, that this time holds for both the perTimeSlot and the perDayAndTimeSlot approach, is that the most useful rules have high support (higher than 0.2 for the perTimeSlot approach, higher than 0.3 for the perDayAndTimeSlot), so they are not filtered by the framework when the minimum support is increased.

To confirm this, it is possible to notice that the trend found in the number of fixed stations is completely different from the one of the number of fixed stations per movement, represented at Figure 5.16.

In this case, it is possible to notice that an higher support implies an higher number of stations fixed per movement. This phenomenon occurs for two reasons:

- Main reason: as stated before, an higher minimum support forces the framework to use only the most frequent rules, which are also the most reliable, being present in an higher number of different timestamps.

- Side effect: it is less likely that changing the occupancy rate of the stations in a neighbourhood can produce other critical stations in adjacent neighbourhoods,

since a lower number of rules implies a lower number of addressed neighbourhoods, hence a lower number of potentially critical adjacent neighbourhoods.

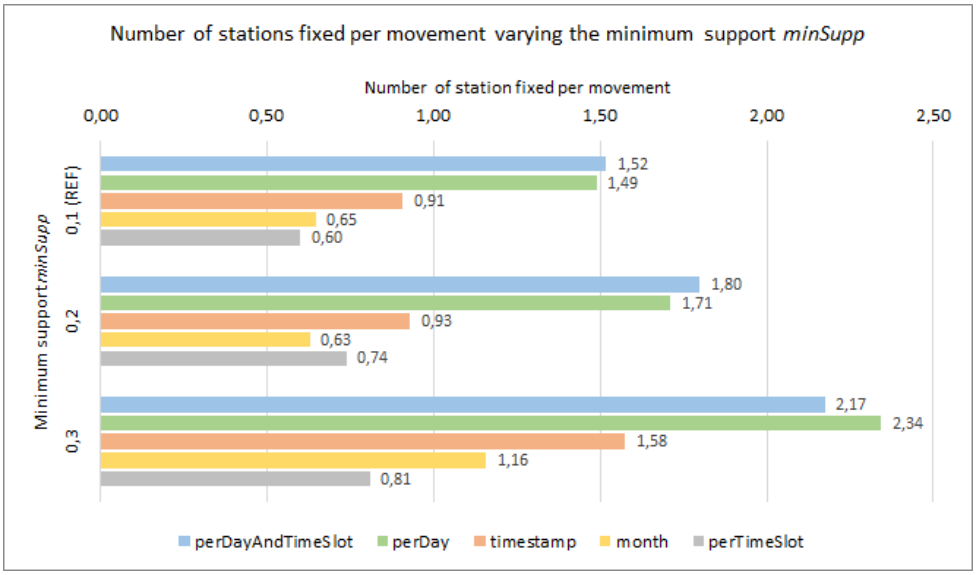

Figure 5.16. Number of stations fixed per movement for different minimum supports.

Looking at this graph it is important to notice that the perDay and the perDayAndTimeSlot are the most effective approaches to rebalance the system, having comparable performance, while all the other approaches are far behind.

Indeed, even if the increment in the framework's accuracy is significant (respectively +14.7% and +57% for support equal to 0.2 and 0.3 over the reference configuration), it was decided to give priority to the absolute number of fixed stations, since a difference of 47 (-21%) and 82 (-36.5%) fixed stations, respectively for support 0.2 and 0.3, is not negligible and would substantially degrade the quality of the offered service.

5.3.5 Effect of number of trucks *N*

The last parameter that needs to be addressed is the number of trucks N. As explained before, this parameter affects the number of neighbourhoods that the framework is able to rebalance in a given repositioning hour, since it was supposed that, at every repositioning hour, a single truck could rebalance at most the stations in one neighbourhood. As a consequence, for a very large city such as Barcelona, it was decided to apply the rebalancing for N equal to 5, 10 and 20.
The results of this analysis are displayed in Figure 5.17, in terms of number of stations fixed by the different configurations, and in Figure 5.18, in terms of number of stations fixed per movement.

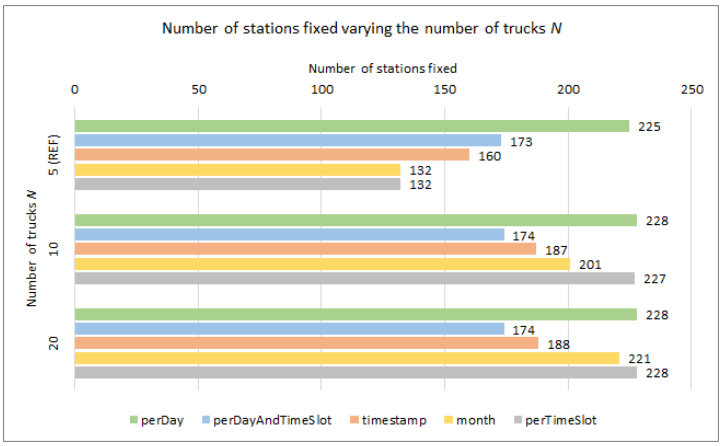

Figure 5.17. Number of stations fixed using different values for the parameter N.

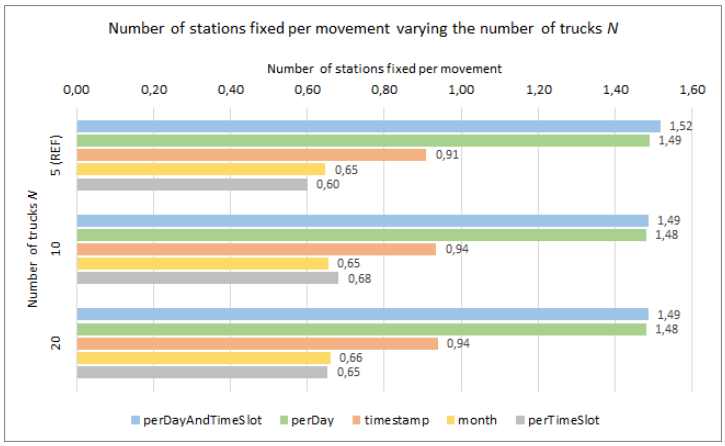

Figure 5.18. Number of stations fixed per movement using different values for the parameter N.

These charts provide many important insights on the framework performance (and, indirectly, also on the input dataset):

- As stated before, the perDay approach is the best performing one in absolute terms, independently from the value of N.
- The framework is able to fix at most 228 stations. This number depends on the specific problem we choose to solve (hence to the values of neighbourhood radius and critical threshold) and is influenced by Barcelona's dataset, which does not allow the framework to extract many association rules, and by the concept of *critical station* and of *neighbourhood*.
- The perDayAndTimeSlot approach seems to have a more strict upper bound, but this time this behaviour is due to the lower number of rules extracted by this approach with respect to the others. To confirm this, it is possible to look at the number of stations fixed per movement by the perDayAndTimeSlot approach, which is the same both for $N = 10$ and $N = 20$. This phenomenon implies that the framework is applying always the same rules, hence that, even for $N = 10$, the framework is already exploiting all the applicable rules extracted from this approach. It is likely that a cleaner initial dataset would have enabled the framework to extract an higher number of rules for all the approaches, therefore to reach higher performance.
- The absolute performance of the perTimeSlot approach are comparable to the ones of the perDay approach when using 20 trucks. Pay attention, even if the absolute number of stations fixed by these two approaches is the same, it does not mean that they are comparable. To confirm this, it is sufficient to compare the number of stations fixed per movement by these two approaches to understand that, in order to fix the same number of stations of the perDay approach, the perTimeSlot approach has to perform many more movements, 127% more to be exact.
- For the first time, the month approach has better absolute performance than the timestamp one. Also in this case, it is important to look at the number of fixed stations per movement, to understand that the month approach has better absolute performance than the timestamp one only because it applies more rules[10] . To confirm this, the latter is much more effective than the

[10]As a reminder, the month and timestamp approach use the exact same discordant rules, but while the timestamp approach checks at filtering hour which rules are applicable, the month assume that the critical stations at filtering hour will always be similar during the week, hence it filters the applicable rules looking at all the critical stations present at filtering hour in the validation set. As a consequence, the set of rules that are considered applicable at filtering hour

former, having a number of fixed station per movement at least 38% higher in any of the considered configurations. Furthermore, this difference in relative performance highlights that assuming that the usage patterns of the bike sharing system do not change during the week is completely wrong, answering to the question "Does the usage of the bike sharing system vary according to the day of the week?" asked in Section 4.6.3.

Considering all these insights, as stated in Section 5.2.1, it was decided to choose as best configuration the one that uses 5 trucks to rebalance the system, since adding 5 or 15 more trucks would have lead to a negligible increase in performance, while respectively doubling or quadrupling the costs related to the trucks.

by the month approach is a super-set of the rules applicable at filtering hour for the timestamp approach.

Chapter 6

Conclusions

In conclusion, it is possible to say that the objective of this thesis work, namely developing a framework offering dynamic system rebalancing for a generic target system s, was more than achieved.

The presented framework offers five possible rebalancing approaches to rebalancing a target system. Furthermore, it is both extremely versatile, being managed through 9 different input parameters that can be tailored to any target system, and extraordinarily modular, since each framework sub-block can be executed as if it was a standalone project.

The performance that the framework was able to achieve are more than satisfactory, considering that Barcelona's dataset was very challenging and needed a lot of preprocessing in order to extract knowledge suitable to rebalance the bike sharing system. Moreover, it is important to remember that the presented framework has the purpose of rebalancing a real bike sharing system and was designed to understand the usage patterns of Barcelona's citizens and to rebalance the system, allowing it to match the end users requests during the whole day.
Unfortunately, in the context of this thesis, it was not possible to apply the framework to Barcelona's bike sharing system, so, in absence of better options, it was decided to measure the performance of the framework just after the rebalancing took place. As a consequence, it was not possible to show the real performance of the developed system, because this validation approach did not highlight the long term effects of the rebalancing, which were the real target of the framework.

It could be very interesting to apply the framework on cleaner datasets produced by different cities, in order to understand how much the quality of the data and the topology of the target city affect the rebalancing performance. Furthermore, in order to measure the real capabilities of the framework, it should be tested on

the bike sharing system of an existing city, observing how the system evolves in the next few hours after the actual rebalancing took place.

Finally, given the very limited literature on dynamic system rebalancing, both for general systems and for bike sharing, it is reasonable to say that the natural continuation of this thesis work is to refine the framework even more and compare its performance to the state of the art, in order to understand if this new, totally data-driven approach, is able to outperform the existing rebalancing algorithms applied to bike sharing and, more in general, if it is able to bring something new to the scientific community.

Bibliography

[1] E. Harper. Can big data transform electronic health records into learning health systems? *Stud Health Technol Inform*, 201:470–475, 2014.

[2] Ishwarappa and J. Anuradha. A brief introduction on big data 5vs characteristics and hadoop technology. *Procedia Computer Science*, 48:319–324, 2015.

[3] Josh Constine. How big is facebook's data? 2.5 billion pieces of content and 500+ terabytes ingested every day. https://tcrn.ch/NhjAVy, 2012.

[4] Pang-Ning Tan, Michael Steinbach, Anuj Karpatne, and Vipin Kumar. *Introduction to Data Mining (2nd Edition)*. Pearson, 2013.

[5] Complete guide to association rules. https://towardsdatascience.com/complete-guide-to-association-rules-2-2-c92072b56c84.

[6] Tomas Borovicka, Marcel Jirina, Pavel Kordik, and Marcel Jiri. Selecting representative data sets. In *Advances in Data Mining Knowledge Discovery and Applications*. InTech, September 2012.

[7] Spark 2.2.0 documentation: Machine learning library (mllib) guide. https://spark.apache.org/docs/2.2.0/ml-guide.html.

[8] Spark 3.0.1 documentation: Spark overview. https://spark.apache.org/docs/3.0.1/.

[9] Spark 2.2.0 documentation: Frequent pattern mining. https://spark.apache.org/docs/2.2.0/ml-frequent-pattern-mining.html#fp-growth.

[10] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. *SIGMOD Rec.*, 29(2):1–12, May 2000.

[11] Haoyuan Li, Yi Wang, Dong Zhang, Ming Zhang, and Edward Y. Chang. Pfp: Parallel fp-growth for query recommendation. In *Proceedings of the 2008 ACM Conference on Recommender Systems*, RecSys '08, page 107–114, New York, NY, USA, 2008. Association for Computing Machinery.

[12] Ahmadreza Faghih-Imani, Robert Hampshire, Lavanya Marla, and Naveen Eluru. An empirical analysis of bike sharing usage and rebalancing: Evidence from barcelona and seville. *Transportation Research Part A: Policy and Practice*, 97:177–191, March 2017.

[13] Pierre Hulot, Daniel Aloise, and Sanjay Dominik Jena. Towards station-level

demand prediction for effective rebalancing in bike-sharing systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '18, page 378–386, New York, NY, USA, 2018. Association for Computing Machinery.

[14] Christine Fricker and Nicolas Gast. Incentives and redistribution in homogeneous bike-sharing systems with stations of finite capacity. *EURO Journal on Transportation and Logistics*, 5(3):261–291, June 2014.

[15] Daniel Chemla, Frédéric Meunier, and Roberto Wolfler Calvo. Bike sharing systems: Solving the static rebalancing problem. *Discrete Optimization*, 10(2):120–146, May 2013.

[16] Fábio Cruz, Anand Subramanian, Bruno P. Bruck, and Manuel Iori. A heuristic algorithm for a single vehicle static bike sharing rebalancing problem. *Computers & Operations Research*, 79:19–33, March 2017.

[17] Mauro Dell'Amico, Eleni Hadjicostantinou, Manuel Iori, and Stefano Novellani. The bike sharing rebalancing problem: Mathematical formulations and benchmark instances. *Omega*, 45:7–19, June 2014.

[18] Federico Chiariotti, Chiara Pielli, Andrea Zanella, and Michele Zorzi. A dynamic approach to rebalancing bike-sharing systems. *Sensors*, 18(2):512, February 2018.

[19] Y. Shafranovich. Common format and mime type for comma-separated values (csv) files. RFC 4180, RFC Editor, 10 2005.

[20] Spark 3.0.1 documentation: Rdd programming guide. https://spark.apache.org/docs/3.0.1/rdd-programming-guide.html#resilient-distributed-datasets-rdds.

[21] About train, validation and test sets in machine learning. https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7.

[22] Mikhail J. Atallah and Marina Blanton. *Algorithms and Theory of Computation Handbook*. Chapman & Hall/CRC, 2nd edition, 2009.

[23] pandas documentation. https://pandas.pydata.org/docs/index.html.

Ringraziamenti

Caro lettore,

mi permetto di darti del tu dopo 75 pagine insieme, perché ormai, volente o nolente, conosci una parte di me e di come lavoro.

Sia che tu faccia parte delle persone che sto per ringraziare esplicitamente, sia che tu abbia trovato questa tesi nella Biblioteca del Politecnico di Torino e non abbia la più pallida idea di chi io sia o di cosa io mi occupi, voglio comunque ringraziarti per aver utilizzato il tuo tempo per leggere questo elaborato, a cui ho lavorato assiduamente negli scorsi mesi.

La prima persona che mi sento di ringraziare è sicuramente il mio relatore, il Professor Paolo Garza, che mi è stato vicino per tutto questo tempo e mi ha aiutato a plasmare questo lavoro in ogni sua parte. La sua cortesia, disponibilità e incredibile preparazione mi sono state indispensabili per realizzare questa tesi.

Allo stesso modo, voglio ringraziare la mia co-relatrice, la Professoressa Sara Comai, per aver accettato questo incarico e per i preziosi feedback che mi ha fornito per migliorare questo elaborato.

Voglio ringraziare con la stessa intensità la mia famiglia, che non ha mai smesso di supportarmi durante questi anni, sia emotivamente che economicamente. Senza di loro, niente di ciò che ho fatto sarebbe stato possibile. Un pensiero speciale va a mio cugino Fabrizio, che per ben 3 anni è stato mio coinquilino e che mi ha aiutato innumerevoli volte a risolvere problemi di geometria tanto quanto di cucina.

Vorrei inoltre ringraziare tutti i miei amici, a distanza e non, che mi sono stati vicini durante tutto il corso dei miei studi e che mi hanno aiutato a non mollare mai la presa, neanche nei momenti più difficili. Una menzione speciale va ai miei colleghi facenti parte di quella che scherzosamente abbiamo iniziato a chiamare la *SbadaGang*: Claudio, Walter e Giacomo. Siete sempre stati un'inestimabile fonte di informazioni utili (e non) e sono certo che continuerete ad esserlo in futuro. Vi auguro il meglio, ovunque le vostre vite vi porteranno.

Infine, vorrei ringraziare una persona speciale, che in un primo periodo di questo percorso è stata *solo* un'amica, ma poi è diventata molto di più. Nico, grazie per essere sempre stata al mio fianco e per avermi aiutato durante questi anni, per aver condiviso con me sia le gioie che i dolori e per essere sempre riuscita a farmi sorridere, nonostante la distanza che ci separava. Sei stata fondamentale durante tutto questo percorso e sono sicuro che continuerai ad esserlo in futuro.