POLITECNICO DI TORINO

Facoltà di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea

Predizione di eventi georeferenziati

Relatore
prof. Paolo Garza

Candidato
Giulio Cataldi

Dicembre 2019

Sommario

Ringraziamenti

Ringrazio calorosamente tutti coloro che mi sono stati vicino in questa avventura accademica, fornendomi il supporto necessario per portare a termine un'impresa molto importante per il mio futuro. In particolare, tengo a ringraziare di cuore mia madre che mi ha permesso di raggiungere questo traguardo.
Ringrazio affettuosamente i coinquilini, i compagni di corso e tutti coloro che ho incontrato sul mio percorso: sostenendoci a vicenda siamo riusciti nel nostro intento.

Indice

Sommario

Introduction

Georeferenced data analysis

Nowadays the quantity of data used in computer systems, transmitted on the network and received on our devices is increasing exponentially. Our society is becoming more and more data-hungry and the ability to process huge amount of data is becoming more valuable every day.
Let us think about social networks, ad banners and e-commerce websites. They are all based on the data we daily sent over the network. Systems that analyse our data are placed everywhere around us, and they became integrant part of our life, and more often than not, we do not even notice this.

Today, thanks to the ever-evolving technologies, we have the possibility to store and process enormous amount of data in a relatively easy and cheap way. Analysing the past can help us predict the future, and this is one of the most important challenges of this decade. This is what is known as “Classification”.

A lot of different classification algorithms already exist. They are different among each other for efficiency, accuracy, areas of application and so on. Each and every one of them has its own strengths and weaknesses, and they can (and must) complement each other.

The challenge we will face in this thesis is a classification problem, relative to space and time referenced events both from a general perspective and in a particular case of study.
The objective of this document is to find a new way to classify and predict events of this type, such that both computational efficiency and labelling quality can increase.
For this reason, it was first necessary to do some research about the already existing methods, trying to understand what parts of them could be useful for my purposes, and what aspects should, instead, be changed and improved.
Thereafter, I needed to find a way to analyse data of this type, to correctly process the information it carried and to be able to write good quality code that could build a classification model based on it. In this work, I used the tools I was more familiar with, which are Java and Apache Spark. I will discuss the algorithm from a general perspective at first, going into details later, to give a better explanation of which issues I had to face, and what I decided to do to solve them.

It is important to point out that I did not build anything from scratch. It was more a process of improving the existing tools, adding features that could refine the mined rule and that could avoid some issues that could worsen the efficiency of the process.
The analysis, eventually, will be accompanied and completed by the data collected in testing phase, with the relative conclusions I derived from them, and advices on how to move on from this point.

This is a popular argument, that has been (and is being) faced by many, and for this reason I had the chance to confront my proposed solution with many others. It was interesting and satisfying to find out that what we are going to discuss in this document has different positive points, that made this approach worth a deepening.

The interest for this problem has arisen during my study period. I noticed how this discipline is actually very transversal, and has application in every area of study, work and daily life. Being interested in cybersecurity, and in particular in the prevention or early detection of attacks, it was natural to get to the discussion of this argument.

In the end, while the algorithm has some evident flaws that will require further discussion and proper solution in the future, to make it viable for its purposes, the path we have tracked seems to be promising, and it can lead to even more interesting results.

Chapter 1

Classification: state-of-the-art

1.1 The problem of classification

Classification is a problem that consists in predicting labels for a given set of "unclassified data". In other words, it is the definition of an interpretable model for a specific phenomenon, that allows us to perform prediction on how things will go.
Given a set of training data, in fact, we should be able to train a classifier (which means to create a model), and then to use it to classify as correctly as possible other data.

There are plenty of classification methods nowadays, that differs for their best area of interest. The main ones are Decision Trees, Bayesian Classification, Rule-Based Classification, SVM, Neural Networks and so on.
Each one of them differs for accuracy, scalability, interpretability and has an area of application where it shines for its performances.
As we said, classification has become part of our daily life, since it is used to know our preferences, to protect our ICT systems from cyberattacks, to improve medical diagnosis, and much more.

1.2 Definition of classification

Even if there are multiple ways to perform classification, in the end, we can give a general definition.
Classification is a process that, given a set of class labels and a set of data described with those labels, finds a descriptive profile of each class, so that it is as easy as possible to assign labels to new unclassified data. We call the first set of data (the one already labelled) "training set", since it is used to create the descriptive model, while the second set is called "test set", since it is used to check the quality of the classifier.
The quality is measured with a number of different parameters, that gives information about different aspects of the classifier:

1) Accuracy, which is how many labels are correctly assigned to test data;
2) Efficiency, that gives information on how fast the model is built and the labels are assigned;
3) Scalability, very important in big data, is an index that tells if a model manages to classify huge data sets;
4) Robustness, which is the resilience of the model to noise and missing data;
5) Interpretability, which is a measure of how readable is the model.

1.3 Classification: state-of-the-art

We already said that classification is assuming a more and more important role in a lot of different disciplines nowadays. For this reason, multiple ways to classify data are being used in every aspect of our life.
Each method has its flaws and its strengths. In this subchapter we are going to skim through the various approaches existing today.

1.3.1 Decision trees

A decision tree is a decision support tool that uses a tree-like graph or model of decisions and their possible consequences, including chance event outcomes, resource costs, and utility. In practice, a decision tree consists of a series of conditional statements that are used to understand which class fits best a piece of unlabelled data.

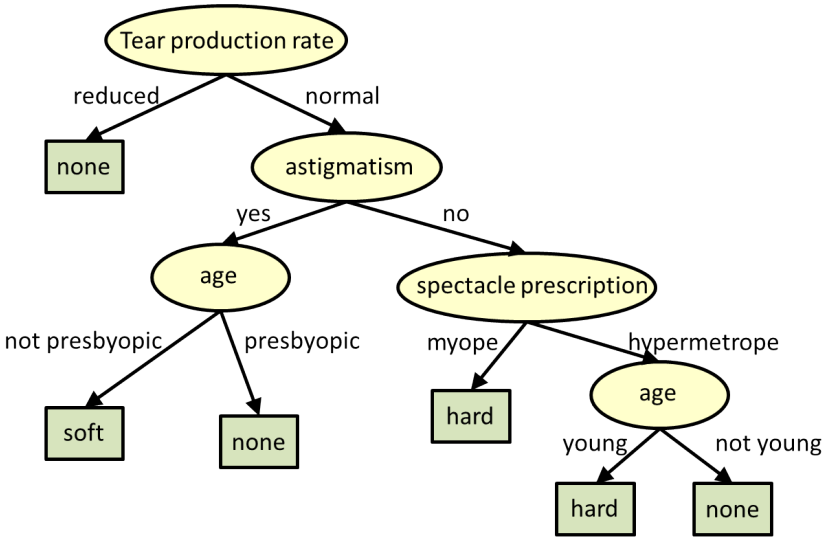

Figure 1.1 Example of Decision Tree

Using a decision tree comes down to starting from the root node, passing the various statements until a leaf node is reached. The label contained in that leaf node is considered the most fitting for that data, and used to classify it.

1.3.1.1 Building decision trees

The main challenge of decision trees is to find the best way to build them. Currently, there are many algorithms used for this purpose; an example of these is Hunt's Algorithm, which is one of the oldest, too.

The way Hunt's algorithm builds a decision tree is pretty simple. Given a training data set, it recursively checks which attribute is the local best to perform a split. Based on that attribute, the data set is split into 2 or more data sets, and the procedure goes on for each one of them. When a data set contains records all belonging to the same class, the procedure for that subset stops, and a leaf node with that class label is created. If the data set is empty, a leaf node with the default class label is created, instead.

Of course, there are some issues with this procedure: the procedure is greedy, since it selects the best attribute looking only at what is better in that moment, but not considering what could be the global optimum.
Moreover, the structure of the test condition has to be discussed too, since it can be binary (the subset is divided in two), or multiple (the subset is divided in any number of splits), and it may refer to different types of attributes (numeral, ordinal, continuous).
Splitting numeral and ordinal attributes can be easy, while continuous attributes need some processing: we can discretize them (making the attribute ordinal), or perform a binary decision (is it greater or less than a threshold value?).

Another important issue is to decide which attribute is the best one to perform the split at every step of the training process. It is usually preferred to split on the most homogeneous attribute, but a way to compute homogeneity of nodes is required. At the state of the art, there are different ways to calculate this parameter, each one used by different algorithms, based on which works best for them.
The most popular ways to measure homogeneity are the GINI Index, Entropy and misclassification error.
GINI is and index that measure probability of a node to become part of one of the splits; usually, a pondered average is performed to understand which attribute is the best to perform the split.
Entropy is a measure very similar to GINI, with the addition of a logarithm, and it gives higher values when the split is more homogeneous.
Lastly, misclassification error is a simpler way to decide which attribute is best. It measures the misclassification error made by a node, and it is minimum when the split is completely asymmetric (all records belonging to one of the subnodes), while it is maximum when we have a perfectly symmetric split (50:50).

We also need a stopping criterion when building a tree, to avoid wasting time expanding nodes that are already good enough. The main stopping condition is when a data set has records all belonging to the same class, but if after a given number of iterations, we do not manage to achieve such condition, we can stop when the records have very similar attributes. It is, at the end of the day, important to perform some pre-pruning and post-pruning, to avoid over-computation.

1.3.1.2 Overfitting

Another big problem to address is overfitting. While underfitting stems from the usage of a predictive model that is not adequate for the data set (for example, using a linear model for non-linear data), overfitting happens when in training phase noises or residual variations are considered as if they were descriptive of the underlying model. For this reason, an overfitted model loses the ability to generalize the phenomenon, becoming useless in performing predictions on data different from the training set.

To address overfitting, we use an Early Stopping Rule (pre-pruning), stopping the training phase before the model becomes too particular. We can define a various number of stopping rules (data belonging to the same class, attributes being similar, number of instances being less than a user-defined threshold…).
There is also the possibility to perform post-pruning, allowing the model to become a fully-grown tree, and then trimming the nodes from the bottom to the top. If we notice that the generalization error improves after trimming, we proceed in this way.

1.3.1.3 Other issues

The usage of a decision tree brings other important problems:

1) Data fragmentation, when the number of instances on lower nodes is so small that it loses any statistical meaning;
2) Missing values, that negatively affect the building of the tree, since it impacts on the impurity indexes, on the testing of data with the missing value and so on;
3) Search strategy, since building the optimal tree is a NP problem, and for this reason we have been using greedy algorithms.
4) Expressiveness, because there are different problems that cannot be easily modelled with decision trees (e.g. continuous variables).

1.3.2 Rule-based classification

The rule-based classification is a process of data analysis in which unlabelled data is classified by means of a set of conditional statements. Each statement has an antecedent,

which gives the requirements to be accepted in that class, and a consequent, containing the label of the class. An example is the following:

```
(State influences economy)^(high tax rate)=>communism

(Economy is independent)^(medium tax rate)=>libertarianism
```

The LHS is the antecedent, or rule condition.
The RHS is the consequent, and it is the label.

We say that a generic rule R covers an instance X if and only if X satisfies the conditions imposed by R. In the previous example, USA satisfy the second rule and it is classified as a libertarian country. China, on the other hand, satisfies the first rule, and it is put in the communism group.

1.3.2.1 Characteristics

Rules has to be mutually exclusive, so that we cannot apply more than one rule for a given instance (we want to avoid ambiguity).
The rule model must also be exhaustive, taking into account every possible combination of rules, so that each record can be properly classified.

We can see how rule-based classification is centred around conditional statements, just like decision trees. In fact, it is possible to pass from decision trees to rules, by writing a model with the statements contained in the various nodes of the tree.
Furthermore, it is possible to simplify the rules so that the process of classification becomes faster.
The problem with simplification is that it may cause the loss of mutual exclusiveness and exhaustiveness.

To overcome the lack of mutual exclusiveness, it is possible to order the set of rules by priority, and perform the classification by scanning down the set, and selecting the first rule that is verified.
Instead, to address the problem of exhaustiveness, there is the possibility to introduce a default class, so that if an instance do not meet the requirements of any rule, it can be assigned to that class.

Building classification rules can be done directly, by analysing the training data and finding frequent patterns, or indirectly, from other previously built models (like we just saw with decision trees).

Using classification models has a lot of advantages: they are fast and easy both to build and to test, they are as expressive as decision trees, and they are very readable.

1.3.3 Associative classification

As we just saw, using rules to performs classification is easy and gives high quality results. Rules are based on a series of conditions that we use to decide if an instance is part of that class. But if instead of looking at high-level conditions, in the antecedent we place itemsets, we obtain the so-called association rules.
An association rule is a particular rule that states that if we find a given set of items in a transaction, then, we can predict an event Y.

```
(x, z, a) ==> y
```

The rule above states that if we find items x, z and a together, then y will be true.

The strengths of such a model are that it is easily interpretable, scalable and efficient for classification. Moreover, it is more accurate than decision trees, and it is not affected by missing data.
However, if the support threshold is too small, the rule generation may become excessively slow.

1.3.3.1 Association rules mining

The objective of algorithms based on association rules mining is to analyse a transactional database and to find out frequent itemsets (or patterns). For example, a website of e-commerce may analyse the products that are usually looked for in the same session of navigation on the site, to extract frequent patterns, and be able to advice customers on what to look next. An example can clarify the concept. If the data analyst of said website finds out that when a person looks for a CPU and a RAM, 80% of the times, it also looks for a motherboard, than he can extract a rule like this:

```
(ram, cpu) ==> mobo
```

and use it in the future to redirect people looking for RAMs and CPUs to insertions about motherboards.

1.3.3.2 Definitions

Before describing the most popular algorithms for association rules mining, we need some basilar definitions:
Itemset: it is a set of items in the transactional database;

Support of an itemset: it is the ratio between the number of transactions containing that itemset and the number of total transactions in the database. It is a measure of frequency of the itemset;
Frequent itemset: it is an itemset whose support is higher than a given threshold (minimum support);
Confidence: it is the ration between the number of transactions containing a given itemset and the number of transactions containing an itemset that has inside the previous one.

1.3.3.3 Association rules extraction

It is the process to build a classification model based on association rules, starting from a transactional set of data.
The process of extractions scans the transactions, and extracts the rules with support and confidence both greater than the given thresholds.
The result is both complete and correct, since the model will contain all and only the rules satisfying both constrains. Of course, further constraints may be added, to obtain a better model.

Brute-force approach
The most simple way to extract rules from a transactional database is to list all the possible itemsets, compute their support and confidence, and keep only rules that satisfy both constraints.
This may be doable for small data sets (which, by the way, are not very interesting), but has no scalability, and it is almost always unfeasible.

The association rule extraction process may be split into two subphases: extraction of frequent itemsets, and then extraction of rules from the itemsets.
The first step is the most computationally expensive, since it requires the generation of all possible permutations of items in the database.
The second step consists in trying all possible binary partition of the frequent itemsets, enforcing the confidence threshold.

If we apply the brute force approach on a data set containing T transaction of length l, with d possible different items, the complexity of generating all itemsets is

$$O(T*2^{d}*l)$$

which is practically never feasible.
To improve the efficiency, we must reduce these 3 parameters somehow.

1.3.3.4 The Apriori algorithm

To make the itemset extraction more efficient, we can use the simple Apriori principle, that states:

"If an itemset is frequent, then all of its subset must be frequent, too"

This principle is due to the anti-monotone property of support. In fact, if we have two itemsets A and B, and A is contained in B, then the support of A is greater than or equal to the support of B. If we invert this statement, we can use the Apriori principle to perform pruning while generating transaction.

In fact, if a set is found to be infrequent, than all sets containing that set must be infrequent too.

The Apriori Algorithm is an approach based on levels, since at each step it analyses itemset of the same length.

C_k: Candidate item set of size k
L_k : Frequent item set of size k
L_1= {frequent items};
For (k=1; L_k!=Φ;k++) do begin
C_{k+1} = candidates generated from L_k;
For each transaction *t* in database do
Increment the count of all candidates in C_{k+1}
Those are contained in *t*
L_{k+1} = candidates in C_{k+1} with min_support
End
Return U_k L_k;

Apriori pseudo-code

For each level, it executes two steps: first, the candidate generation of length k+1 by joining frequent k-itemsets, and relative pruning by applying the Apriori principle (discards k+1 itemsets that contain at least one not frequent k-itemset), second, the frequent itemset generation, scanning the dataset to compute the support of each k+1-itemset and pruning those who do not respect the support threshold.

Apriori algorithm has for sure its historical importance, but it suffers of detrimental performance issues.

Firstly, the candidate generation is a very heavy step, since the candidate set is likely to be enormous. In particular, generating the 2-itemsets is the most critical part. Furthermore, extracting all frequent itemsets requires the generation of all subsets.
Secondly, Apriori performs too many database scans: if the longest frequent itemset is long n, we need n+1 scans.

In the end, the factors that influence Apriori performances are: the minimum support threshold (lowering it greatly increases the number of candidates), the dimension of the data set (more space required to count the support of each itemset), size of the database (since Apriori requires multiple scans of it), and average transaction width.

1.3.3.5 FP-Growth algorithm

The FP-Growth algorithm was designed to overcome the Apriori limitations. In particular, FP-Growth exploits a compressed memory representations of the database, named FP-Tree.
It is characterized by a high rate of compression for dense data distributions (in case of sparse distribution, the compression is lower), and it is capable of completely representing the dataset, so to correctly enforce the support threshold.

The main strength point of this algorithm is that it requires only two scans of the database: the first one is needed to compute the support of the items, and the second to build the FP-Tree.
At that point, the algorithm is a recursive visit of the Tree, with a divide-and-conquer approach.

FP-Tree construction
To build this data structure:

1) Count the support of each itemset and prune those below the threshold;
2) Build the so-called Header Table, by sorting itemsets in decreasing support order;
3) Create the FP-Tree by scanning again the Database: for each transaction in the Database, sort its item by decreasing support and place the transaction in the FP-Tree, following the existing branches when possible, or creating new ones.

FP-Growth application
Once the tree has been built, the algorithm does not need to scan the Database anymore. The frequent itemsets will be extracted in a recursive way from the tree.
The algorithm scans the header table from the item with the lowest support, to the one with the highest support, and builds the header table relative to that item, calling the FP-Growth algorithm recursively on it.

1.3.3.6 Other approaches

Apriori and FP-Growth are the most famous algorithms for association rules mining, but there are many other ways to perform the extraction.

In particular, many approaches to this problem address the issue of data representation, either verticalizing the data layout, or using a compact representation.

In other cases, the usage of the concept of maximal frequent itemset may be exploited: an itemset is said to be maximal frequent if it has no immediate successors that are frequent.

In every approach we are going to use, by the way, we must dimension correctly the minimum support threshold: if it is too high, there is the chance to lose interesting infrequent items, while if it is too low, the problem is likely to explode, and will not lead us to any results.

1.4 Conclusions

Nowadays we have a many ways to perform classification and to try and predict events. The biggest challenge is to find the best approach possible for each problem, in order to compute efficiently the model and to perform correctly the predictions.
From this point, my work started, and we decided to use a different approach to solve our classification problem. In the next chapter I am going to explain why we thought it was necessary to introduce a new way to classify events.
I will show what type of problem we tackled, analysing the issues stemming from it, the necessary actions to solve them, and the way to test the proposed solution.

Chapter 2

Problem statement and description of the proposed prediction technique

2.1 General problem and solution

In this chapter I am going to describe the general issue we wanted to tackle with this work, and the solution I was able to provide in order to obtain better results.
At first, we will give a glance from a general perspective, pointing out only the main aspects of the general scenario, of my work and of the solution, while, later, I will explain every part and every step more in detail, so that everything can be clearer.
I will use some pseudo-code to describe properly what issue I am tackling at every step, and what I am doing to solve it, and offering the relative explanation to it.

2.1.1 The problem

Let us first introduce the problem.
We are given a set of events, that happened in different places at different times. These events have been stored in a given format in a big file. Our purpose is to analyse the data of that file, trying to find out if there are relationships among those events, and if we can find patterns that repeat themselves.
It would be interesting to find out that these patterns are actually independent from where and when the events take place. What I mean by this is that if 3 events occur with a given difference of time among them, in 3 places that have a certain relative position in the space, I want to find out if those events happen with the same differences of time in 3 other places that have a similar space relative disposition.
In fact, it would be interesting to be able to make predictions in the future. This could be used in many different situations: predicting which road will have the best traffic condition, improving weather forecasts, preventing cyber-attacks and so on.

What we are going to do is to extract space-time patterns from past events and use them to create a predictor. This, at the end of the day, means that we are going to train a classifier, test it and eventually use it, if the results are better than what we have at the state-of-the-art.

2.1.2 Proposed solution

Since the classic methods reported in the previous chapter did not lead to satisfying results, I tried exploring a new way, and getting to a new approach.
The main issue that those algorithms have is that they do not consider the space-time relationships among events.
The solution I am about to propose starts from this very consideration, and by performing proper operations on the given data, it is able to improve the quality of classification.

I am going to use the notorious FP Growth algorithm for the association rule mining, since it is better than the Apriori one (both computational and performance-wise), and Spark provides an efficient implementation of it.
By the way, just applying FP Growth on the data set I am given is not enough. I need to perform some operations in order to have higher quality rules.

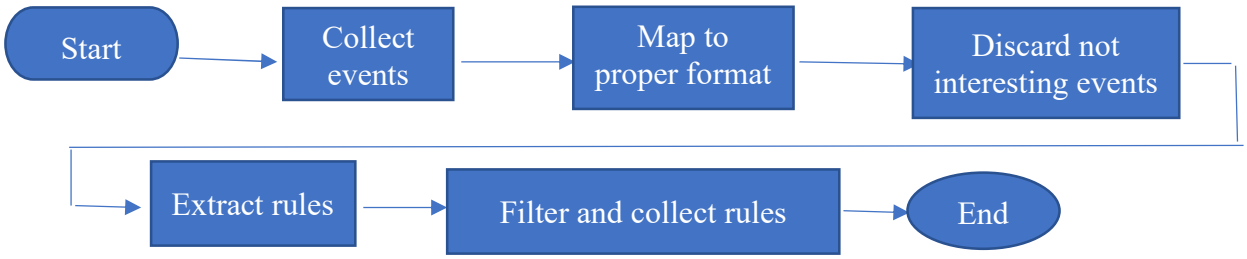

Figure 2.1 Logic flow of rule mining

Let us discuss what the process of rule mining will be like.
As we can see from Figure 2.1, we will start by collecting data. We are expecting these data to be a set of events with information about where and when they took place. Of course, we are also supposing these events to be consistent.
We will, then, map and filter these events. The former operation is needed because instead of having information about time in an absolute form (e.g. a date), we would like to have something that tells us the relative time distances among them. We define the time unit, and we measure the deltas among the events.
The filter, on the other hand, is required because not all the events are going to be meaningful for our classifier. In this way we remove non-necessary information, to make the process less complex and lighter.

The next move is to create temporal windows of events. A window is a collection of events that are temporally near. For example, a window of dimension 4 will contain events that are no more than 4-time units distant. This is very important to keep the information about time consistent while performing association rules mining.
Then, we are ready to extract rules from these windows, and to collect them so that we can use them to perform classification.
Before collection, though, we filter out those rules that are superfluous or not meaningful. Again, this is a way to speed up the classifier and to make it less computational-heavy.

At the end of all this procedure, we will have a set of association rules, telling us what space-time patterns are found to be frequent in the data set. This, in other words, means that we have successfully trained our classifier (or predictor), and that we can finally use it.

In this work, I stressed the classifier in many different ways, so to understand what its strengths and weaknesses were.
To test its functionality, we will use a data set similar to the one used to train it, and we will operate in different ways, introducing constraints and modifying parameters.

The first two parameters that are interesting to monitor are minimum support and confidence of the extracted rules and see how the behaviour of the classifier changes accordingly. I want to tune these two values (that are user-defined for this reason) until I find an optimal trade-off between execution time and quality of predictions. As a trend, I will try to go down with the minimum support, so that I can consider as more events as possible, while using a high value of confidence to extract only meaningful rules.

I also want to see if introducing some constraints helps the classifier performing better. In general, what I want to do is to use some parameters to discard events that probably do not influence other events. In the particular case of study we are going to discuss, this will be clearer, but generally speaking, this means that if I think that two events are too far away in time, in distance or in other ways, then there is a good chance that they are have no influence on each other, and I can make everything simpler by discarding their relationships.

2.2 Implementation of the proposed solution

Now I am going to describe the actual implementation of the general solution reported in the previous part. I have used Apache Spark and Java, because they were the tools I was most familiar with, and they already provide some useful libraries to work with Big Data.
All the considerations we will make here are based on the sample data I used for my experiments. In particular, I used one very big .csv file (named registers.csv) about the states in different periods of time of the bike stations in Barcelona. Each line of this file gives us

the situation of a particular station at a given time of a given day. The format adopted by this file for each line is the following:

```
ID    datetime  freeslots usedslots
```

ID is a positive integer number that identifies the station in the bike sharing system of Barcelona;
datetime gives information about when the measurement was performed, in the format yyyy-MM-dd hh:mm:ss;
freeslots is another integer number telling how many slots were free in that time frame;
usedslots, similarly, tells us how many slots were used at that time.

An example of a line of the register.csv file is the following:

```
1    2008-05-15 12:50:00   7    11
```

The meaning of this line is that at 12:50:00 of 15th May 2008, the station number 1 had 7 free slots and 11 slots occupied with a bicycle.
A thing to be noted is that, even if the file is "comma-separated", the actual separator is a tabulation.

This file actually contains everything we need to perform our considerations and our analysis. In fact, we can exploit it to try and predict future situations, but it requires a process that can be divided into 3 main phases: pre-processing, actual rule extraction, post-processing.
Pre-processing is where we just transform the file in a format that we can use to properly perform rule extraction.
Actual rule extraction is where we create windows of transactions and we apply the Spark-provided methods to extract association rules.
In post-processing we format the rules, sort them and filter them, keeping only the ones that have meaning for our predictions (more on this later).

Given these 3 main phases, we can further divide them into "sub-phases" that describe simple and "atomic" operations on our data.
We can see what these phases are in Table 2.1. In this chapter I will discuss in an analytic way each and every step.

PRE-PROCESSING	Read file
	Extraction of the oldest date
	Sample data
	Filter critical states and set proper format
PROCESSING	Create windows
	Filter windows
	Create transactions
	Apply rule extraction algorithm
POST-PROCESSING	Format rules
	Filter interesting rules
	Store rules

Table 2.1 Steps of rule mining

After this process, we will have our rules, in a usable format and sorted by decreasing rank.

2.3 1st phase: pre-processing

This is the first main phase, made of 4 very simple steps.
The purpose of this initial transformation is to make data usable for our rule extraction.
What we do is to take the original file, store it in main memory, change the format of each line and sample it based on a parameter passed by the user. The sampling is needed because events in the file are too dense: events are distanced only two minutes between each other. This is too complex and of scarce interest, as we will see in the relative subchapter.

2.3.1 Reading file

Of course, the first step is to set the Spark environment, so we can read the file, perform our transformations and actions on RDDs, and apply the rules extraction algorithm.
Then we simply proceed to read the file whose name is passed by the user as the first argument.

```
Create Spark Session;
Create Spark Context from Spark Session;

Create collection of Strings; //each string is a line of
the file
Store the file in the collection;
Cache the collection;
```

We cache this collection because we are going to perform multiple actions on it, and caching is an efficient way to increase performances.

2.3.2 Extraction of oldest date

What we do in this short phase is to extract the record with the oldest date (after having deleted the header record). Why we do this is explained later, when we will actually use this information. We also transform this oldest date into seconds.

```
Create a String; //this will contain the oldest record
Reduce the collection containing the lines of the file;
      For each couple of lines
           Split line1;
           Split line2;
           Compare date1 and date2;
           If date1>date2
                return date2
           Else
                return date1;
      End foreach
Store the result of reduce //it's the oldest date in file
Convert the date to seconds;
Store the result;
```

This oldest date is considered the temporal origin of the data stored in the file (t_0), and all other times will be marked based on their distance from this instant.

2.3.3 Sample data

There is a problem with the registers.csv file: the data collection about every station is performed every minute. Two issues stem from this:

1) Low interest in predicting what will happen in 1/2/3 minute(s);
2) Complexity explosion.

We solve this by sampling the input data. In fact, we do not keep all the records, but filter them by means of a user-defined parameter ("interval"). Let us suppose that the user defines

an interval of 15 minutes. Then we will start from the first datetime and keep only those records whose datetime is distant from the oldest one of a multiple of 15 minutes.
For example, if the oldest time is 15-08-2008 12:00:00, we keep 15-08-2008 12:15:00, 15-08-2008 12:30:00, etc, but we discard 15-08-2008 12:16:00.
There will be a discussion on the best values for this parameter later.

```
Apply sampling filter to input collection;
```

This line of code is used to perform this sampling. The sampling filter takes a user-defined interval value, and uses it as follows (for each record in the collection obtained by the file):

```
SamplingFilter{
 Foreach record:
  Split the record;
  Split the timestamp;
  Convert timestamp to seconds;
  If timestampInSeconds % interval == 0
          Keep the record;
  End if
  Else
          Discard the record;
  End else
 Endforeach;
}
```

At this point we are ready to perform the last step of pre-processing.

2.3.4 Filter critical states and set proper format

We are at the crucial point of this main phase: we are, in fact, ready to remove those records that are not meaningful for our purposes and to format the lines, so they are adequate to be discussed.
In particular, we want to pass from the format described before (the one of the register.csv file), to the following:

```
ID    timestamp      event
```

ID is the same as before, timestamp is a way to represent a delta from instant 0, event is either “Empty_station” or “Full_station”.
To be clearer, here is an example of this new format:

```
1    4    empty_station
```

This record tells us that at time 4 (which means 4 time units after time 0, the oldest timestamp in the whole file), the station number 1 was empty. Note that both the dimension of time unit, and the threshold to decide whether a station is empty or not are passed as arguments by the user.

Another thing to be noted is that not all records of register.csv will be taken in consideration: as we said we are interested only in critical states. We need, then, to remove those lines in which is not represented a critical state, in other words, those records where both the free slots and the used slots are not under the user defined threshold.
Furthermore, the sensor system that collected data about the states of the stations had inevitable problems during the time span in which it worked, and for this reason, there are records where both these two values are 0. We will remove these records too, because they are wrong, and they would bias the rule extraction.

This is how we take only critical states (while also removing bugged lines):

```
Apply "Filter Function" to current collection;
Store filtered collection;

Filter Function{
 Foreach record:
      Split the record;
      Store freeslots and occupiedslots;
      If(freeslots == 0 AND occupiedslots == 0)
           Continue;
      If(freeslots < threshold)
                Return record;
      If(freeslots > threshold)
                Return record;
      Else
                Continue;
 End foreach;
}
```

In this example we see the variable "threshold". This is a value set by the user, to decide when we consider a station in a critical state.

After this we can finally map the remaining lines to the format we want. This is where the previously extracted "first date" (minDateInSecs) plays a fundamental role: to pass from

datetimes to timestamps, we need to perform the difference between the datetime of each line and this “first date”, and then divide it by the user defined “time span”.

```
Apply Map function to the filtered collection;

Map{
 Foreach record:
  Split the record;
  Store ID, Timestamp, freeslots, occupiedslots;
  Convert the timestamp to seconds = timestampsinseconds;
  Instant = timestampinseconds-mintimestamp/interval;
  //mintimestamp is the first instant in the file
 converted to seconds, interval is the user defined value
 of the sampling interval
  If(freeslots < threshold)
           Return “StationID   Instant   Empty_Station”;
  If(occupiedSlots < threshold)
           Return “StationID   Instant   Full_Station”;
 End foreach;
}
```

We finally have the records properly formatted. We can now proceed with the actual rule extraction.

2.4 2nd phase: Association rules extraction

This second main phase is the core of our process. This is where we create windows of transactions using the previously formatted data and use the Spark libraries to perform the rule mining.

Some of the created windows, though, will be filtered. In fact, we can say that if a window does not contain a single transaction that refers to the first instant of that window, the same window is actually a repetition of another one, translated of 1 or more time units. For this reason we can eliminate it and make the computation lighter.

2.4.1 Create windows

To better understand this part, we must first define clearly what a window is.
A window is a group of records (in the format obtained from the pre-processing phase), whose distance between each other is less than or equal to a user-defined interval.
A window refers to a starting timestamp.
Let us give some example to better clarify the concept.

Let us assume that the user defined a window dimension = 3. Then, in the window that has starting timestamp = 5, we will find all the records and only the records that have timestamps 5, 6, 7.
Another way to see this is that, if the window dimension is 3, a record with timestamp x will appear in the windows starting from x-2, x-1 and x.
If we add the information about the interval (also defined by the user, as we have already seen), and assume that it is equal to 30 minutes, a window of dimension 3 will contain all the records that are distant 0, 30 or 60 minutes. This means that the classifier we will obtain will perform predictions on a time span of maximum one hour.

To be even clearer, here is a basic example.
If, after the pre-processing, we obtained these few records

```
1    0    stazione_vuota
1    1    stazione_piena
2    1    stazione_piena
3    2    stazione_vuota
3    3    stazione_vuota
4    0    stazione_piena
```

We will get these windows:

```
(1   0    stazione_vuota), (1   1    stazione_piena), (2 1
stazione_piena), (3   2    stazione_vuota), (4   0
     stazione_piena);

(1   1    stazione_piena), (2   1    stazione_piena), (3   2
     stazione_vuota), (3   3    stazione_vuota);

(3   2    stazione_vuota), (3   3    stazione_vuota);

(3   3    stazione_vuota);
```

The first window starts from timestamp 0, and so it contains records with timestamp = 0,1,2.
The second one starts from timestamp 1, thus it contains records with timestamp = 1,2,3.
The third starts from 2, and it contains records with 2,3,4.
The last one starts from 3, and it contains records with 3,4,5.

There is a last consideration. The format of the windows used above is not 100% correct. What we will actually do when creating windows will be to change timestamps so that the first instant of every window will be 0. We will simply translate every record of the window starting from x of -x. This will make our window filtering and rule extraction much simpler.

```
//for each line, I emit a set of lines, one for each
//window the line is part of
Apply "Line Emitter" function to the collection;

//Then, I group the generated lines for instants: each
//instant will contain all the lines part of the window
//starting from that instant.
Group the records of this new collection by key;
```

This is how the windows are created. In the first line of code, that may seem weird, we are emitting, for each line, a set of records that represent that line in all the windows it can belong to.
To better explain this concept, we use an example. If a line has timestamp 4, and the user set the window dimension to 3, then that line can belong to the windows starting from 2 (with relative timestamp 2), 3 (with relative timestamp 1) and 4 (with relative timestamp 0).
We are using a PairFlatMapToPair, because we are not emitting simple tuples, but pairs in which the first element is an integer representing the starting timestamp of the window, and the second element is the actual record (with timestamp adjusted to be relative to the window, as seen before).

This is the pseudo-code of the call method of the class used to perform the PairFlatMap.

```
For each record:
 Create empty list of records;
 Split the record;
 Store the time instant; //instant
 For ( i = instant-window+1; i<=instant; i++)
           //window is the user defined window dimension
           String = "ID   instant-i event"
           Create Tuple(i, String);
           Add Tuple to list of records;
 Endfor;
 Return list of records;
Endforeach;
```

In this code, we get the timestamp of the record, and we emit pairs like this (assuming window dimension = 3):

```
t-2, (id        2     event)
t-1, (id        1     event)
t,   (id        0     event)
```

The reason of proceeding this way is because we want to group all tuples that are in the same window, which is performed by means of "groupByKey" method, called on the collection generated by the aforementioned pairflatmap.

2.4.2 Filter windows

At this point, we have a collection of pairs containing, for each time instant, all the records belonging to the window starting from that instant.
As we said in 2.3, not all the windows we obtain are useful for our analysis, and since the complexity of this process can easily explode, it is wise to prune the more we can.
At this point we can actually perform some pruning. In particular, if a window has no records whose relative timestamp is equal to 0, it means that we can drop it. In fact, such a window is nothing else than a translated replica of another window. If we kept this replica, we would not get any advantage in extracting rules.

```
//Now I filter: only windows with at least one line
//having t=0 are kept, the rest is discarded
Apply Filter to the collection of grouped tuples;
```

This is the simple instruction that allows us to perform this filtering.
The pseudo-code of the call method of Filter is this:

```
Filter{
      Foreach window{
           Foreach event in the window{
                Split event;
                If "it happens at 0"
                     Return true;
           End foreach;
           Return false;
      End foreach;
}
```

We check if one of the records in the window contains a relative timestamp equal to 0. If we fail the search, we drop the window.

2.4.3 Create transactions

This format is good, but in order to use the library provided by Spark for association rule mining we need to transform records in transactions.
We first need to make our collection of pairs a simple collection of strings, and to do this, we will just map the pairs we have to a string. In fact, we do not need the information about the starting timestamp of each window anymore, so we can just drop it.

```
//I map the remaining lines to the desired format
Apply Map to the filtered collection of windows;
```

The Map class has this specific task, as we can see by analysing its pseudo-code:

```
Map{
 For each window{
      Drop information about initial timestamp;
      Emit the list of events of the window;
 End foreach;
}
```

After this mapping, each window will simply be a concatenation of strings, each of which being a record belonging to the window.

Now we can finally transform these windows of strings into windows of transactions, that is what the Spark algorithm requires to perform the mining.

```
Foreach window:
      split window to single events;
      return a Transaction containing those events;
Endforeach;
```

2.4.4 Apply association rules mining algorithm

This is the core part of the whole process. Thanks to the library provided by Spark, we can simply call some methods as they were black boxes on our transaction data, and obtain an output consisting in a set of frequent itemsets (that in this case have little relevance), and frequent rules, which are the one we are interested in.
We will later use these rules to make predictions and test with a variety of experiments, the goodness of our classifier.

In this step, two other user-defined parameters are used. They are the minimum support for the rules, and the minimum confidence.

The support of a rule denotes the frequency of the rule within transactions. A high value means that the rule involve a great part of database. For our purposes, after making many experiments, I noted that it is better to try and keep this value as low as possible (more on this in later chapters).
The confidence of a rule denotes the percentage of transactions containing A which contain also B. It is an estimation of conditioned probability. On the other hand, this value should be balanced in the upper part of the "spectrum". A good value is 85%, since it allows us to obtain significant rules, without filtering everything.

The following pseudo-code is the what was used to perform rule mining:

```
Create a DataFrame based on the input transactions

Create an FPGrowth object

Set the value of min. support and min. confidence
Extract frequent itemsets and association rules by
invoking the fitting method;

Retrieve the set of frequent itemsets

Retrieve the set of association rules
```

The first thing to do is to put the data of the windows in a data structure that can be used by the algorithm. For this purpose we create a DataFrame by means of the createDataFrame method called on the Spark session object.
Now we are finally ready to create our FPGrowth object (as said in the previous chapter, we will use FP Growth algorithm for our analysis), to set the minimum support and confidence chosen by the user and eventually to call the fitting method which performs the computation. At last, we collect the frequent itemsets and the association rules obtained by the method.

Finally, we have a Dataset of frequent rules. We still have some work to do, though, to make them usable for our predictions. This is what Post-Processing is about.

2.5 3rd phase: Post-Processing

The purpose of this last main phase is primarily to perform some pruning. We already said that the extraction part is complex, but in our analysis, also using rules to make prediction can be computationally heavy. For this reason, it is very important to keep only the meaningful rules, so we avoid wasting precious resources and time.

Furthermore, as we have already seen in the general approach table, we also have to format rules so they are understandable by our algorithm, and we have to store them somehow. These processes all take place in Post-Processing.

2.5.1 Format rules

This step is simple, but very important. Using the rules in the format given by the Spark mining algorithm is pretty uncomfortable. What it does, in fact, is to put both antecedent and consequent in a structure called WrappedArray, and to explicitly print them with the word "WrappedArray" before.
The format I decided to use, instead, is the following:

```
antecedent ==> consequent |confidence|lift
```

As you can see, there are 3 slots: in the first one I put the actual rule, with the consequent separated from the antecedent with an arrow.
In the second slot I put the confidence, that, as we will see, will be used to rank rules.
The third slot is reserved for the lift, which is a second parameter used to rank rules, when the confidence is the same.
This was easy to use both for performing the following operations and to do some debugging.

```
Apply Map function to the mined rules collection;

Map{
      Create String s;
      Append antecedent to s;
      Append consequent to s;
      Append Confidence and Lift to s;
      Return s;
}
```

The mapping is straightforward, but very important for later steps, since we will need the four elements to be easily extractable.

2.5.2 Filter interesting rules

This is the core part of post-processing. As we already said, we do not want to have too many rules, so that testing the predictor is a lighter task.
To achieve this, we will prune rules that are not considered to be relevant.
We decided to remove rules that do not respect at least one of the following constraints:

1) The consequent has more than 1 term;
2) The consequent has a term with timestamp greater than or equal to at least one of the timestamps in the antecedent.

The first constraint is needed because our purpose is to keep analysis as simple as possible, without modifying the results. Now, if in a consequent we have two or more terms, we can drop that rules, making the testing process faster, without losing prediction power. In fact, if we have such a rule, we will also have a set of rules with a single term in the consequent that give us the same information.
For example, if we have extracted this rule:

```
(1,0,stazione_vuota),(2,0,stazione_piena) ==>
(1,1,stazione_piena), (3,1,stazione_vuota)
```

then we would also have the following two rules:

```
(1,0,stazione_vuota),(2,0,stazione_piena) ==>
(1,1,stazione_piena)

(1,0,stazione_vuota),(2,0,stazione_piena) ==>
(3,1,stazione_vuota)
```

and we will be able to perform the same predictions.

The second constraint, instead, addresses another subtle issue. The rules are extracted in a dumb way, of course, because the algorithm does not understand what is the meaning of the transactions. For this reason, it will extract some rules in which the consequent refers to an instant preceding the ones in the antecedent. Making predictions for the past has no interest, and for that, we should drop such rules.
Furthermore, we decided to drop also those rules that performs predictions for the same time instant. We already have sensors to tell us what the state of every station is, so we do not need to try and guess it.
So, to conclude, if a rule has timestamps in the consequent preceding the ones in the antecedent, we discard it.

```
Apply Filter to the formatted rules collection;

Filter{
      Foreach rule{
            Split rule;
            Store antecedent;
            Store consequent;
            If(length of consequent > 1)
```

```
                    //consequent must be at most long 1.
                    Return false;
               Initialize minimum instant of antecedent;
               Initialize max instant of antecedent;
               Initialize min instant of consequent;
               Foreach term of antecedent {
                         if (instant < mincause)
                              mincause = instant;

                         if (instant > maxcause)
                              maxCause = instant;

               Endforeach;

               Foreach term of consequent {

                         if (instant < minEffect)
                              minEffect = instant;
               Endforeach;
                    if((minCause   ==   0)   AND   (maxCause   <
minEffect)) { //if it is not to be pruned, I will add it
to my correct rules
                         return true;
                    }
                    return false;
               EndForeach;
End;
```

This piece of code performs the pruning for both reasons.

2.5.3 Store the rules

This is the trivial part in which we store the information for future uses. At the beginning of my work on this thesis, my only aim was to extract rules, and to manually evaluate them. I did not have to perform any test, or to actually use them for predictions. For this reason, in this first moment I decided to simply print them in files and to store them in my HDFS.

Later, I had to use them for real, so I needed some more processing on the format, and on the sorting. We will see more about this in next parts of this document.

2.6 Conclusions

The whole process I described in this chapter was all about extracting rules. This was a delicate part of my work, that required a lot of corrections in order to achieve proper results.

It is very important to calibrate properly the minimum support and confidence. There are some stations that are often empty, and if the support is too high, they prevail on all the other stations, and we get poor quality rules. On the contrary, it is better to keep the confidence at higher values, so the rules that satisfy the constraints are more meaningful and the prediction quality increases.

There were also many bugs to solve, in particular right before and right after the mining algorithm took place. The format of everything had to be double checked in order not to end up causing run time errors.

By the way, this is only the first part of the process, we still have to discuss the part about how we tested the rules, and how we adjusted the extraction process to perform faster and better.

2.7 Testing rules

Until now we have seen how to extract rules from a file like registers.csv. This is already interesting per se, since manually skimming the obtained rules gives us an idea of frequent patterns about bike stations. But the main purpose of my work was to use the rule mining to try and predict future situations of the stations.
For this reason, I have performed a lot of experiments with the same data set used to extract the rules. In this way I managed to have the highest consistency between the training phase and the testing one.

By the way, performing prediction and measuring their quality was not a simple task, but something that required a relevant amount of work both for writing the testing code and for modifying the previously seen rules extracting code.

In the next two subchapters I will describe my implementation for these two processes.

2.7.1 Predictions and quality test: the first approach

After I managed to extract rules properly, I first manually checked them. I wanted to be sure that those rules were meaningful, before starting to use them on actual data.
In my controls, I looked for their correctness (they had to respect the constraints described at 1.4.2), proper syntax and interesting semantic.

The first two were pretty simple to pull off, and I was able to extract rules that followed those two principles easily, while about semantic, I struggled because there were a lot of uninteresting rules, that could actually worsen the performance of the predictor. We will see more about this later, but as an anticipation, I can show a very intuitive example.
Rules like the following one:

```
(128,0,stazione_vuota) ==> (128,1,stazione_vuota)
```

were very common, but they describe an obvious situation, since at first I was using an interval of time of 2 minutes.

We will see later more examples of bad quality rules and the relative solution to these issues.

It is also important to specify that the rules testing has been done by considering one single station at a time, so not to make the complexity of the process explode. This is done by asking the user to pass a parameter containing the station to be tested, and by performing the analysis only on that station.

Eventually, after many experiments with the size of the window, we came to the conclusion that the best size for our analysis was 3. From this it stems that the system was only able to make predictions about events that took place after 1 or 2 time units.
We had to introduce this limitation, because the data set was extremely large, and windows of greater size were too heavy to be handled in an acceptable time.

Now I will describe the process for testing rules and for storing the results in a readable and easy-to-use way. The table below shows the main steps.

PREPARATION	Filtering rules about chosen station
	Shift rules that have t=1 in the consequent
	Give the rules a usable format
	Collect the formatted rules
	Create windows for testing
TEST	Initialize confusion matrix
	Perform prediction and test quality
	Format and store results

Table 2.2 Steps for testing rules

2.7.1.1 PREPARATION: Filtering rules about chosen station

As we have anticipated, performing predictions and quality test about each and every one of the 284 considered stations was not sustainable. In fact, with such a big data set, we had to keep the complexity under control, and making everything in one single process cost us loss in accuracy of our predictor. For this, I preferred to introduce a user defined parameter that contains the station to analyse:

```
String station = argv[6];
```

In this way it is simple to run parallel computations about different stations, both by manually running the code from command line and by using an automatic script.

This "limitation" was simple to introduce, as shown by the pseudo-code below.

```
//I keep only rules about the station in input
Apply the OnlyStation filter to the rule collection;
```

I just call a filter, using the OnlyStation filter, whose pseudo-code is the following:

```
OnlyStation {

      Split the record;
      Store the rule;
      Split the rule;
      Store the consequent;
      Read the station in the consequent and store it in
myStation;

      if(myStation == consideredStation))
           return true;
      return false;
 }
```

We are interested in the station appearing in the consequent, so we split the rule, split the consequent and check if the station ID is the one for which we will test predictions quality.

2.7.1.2 PREPARATION: Shift rules that have t=1 in the consequent

At this point we only have rules about the station chosen by the user.
To produce correct results, though, we need another transformation. After some runs of the code without what I am about to discuss, I noticed some weird results, and started investigating on what could have been causing that. After some thinking, we understood that

the only rules I was considering when collecting the results were the ones whose consequent referred to timestamp = 2. This will be clearer in the part about collecting results, so I will discuss this more in detail later.
This issue was subtle because, due to the dimension of the data set, the vast majority of rules contained a consequent with timestamp = 2. Nonetheless, I could not ignore the many rules (that had, actually, very high confidence!) whose prediction was for something happening at timestamp = 1. Before applying this simple piece of code, all the missed prediction poisoned my results, since the predictor was actually giving the correct interpretation of data, but in testing phase, it was putting these correct predictions in the wrong cell of the confusion matrix.

To solve this problem, we take a simple action: if we find a rule that has timestamp = 1 in the consequent, we translate said rule of 1 time unit. For example, if we have:

```
(1,0,stazione_piena) ==> (2,1,stazione_piena)
```

after applying this line of code, we will obtain

```
(1,1,stazione_piena) ==> (2,2,stazione_piena)
```

Why this is necessary will be clear when we will talk about collecting test results.
The code that performs this transformation is the following:

```
//I shift those rules that have the consequent to t=1
Apply the shifting Map to the collection of rules;
```

Where the pseudo-code of Shift01Rules is

```
ShiftRules {

 Foreach Rule:
      Split the rule;
      Split the consequent;
      if(time of consequent == 1)
           replace 1's with 2's;
           replace 0's with 1's;
      return shiftedrule;
 Endforeach;

 }
```

As you can see, it is a simple mapping process, in which we perform the substitutions of the timestamps.

2.7.1.3 PREPARATION: Give the rules a usable format

One could think we are now ready to start performing predictions now, since we collected the rules, kept only the ones about my station and made them consistent with each other. In reality, we still require a last effort: given the huge amount of data, when we extracted the rules, for each station we have a lot of them. We have a big set of rules that predict that the station will be empty, and a big set of rules that predict that the station will be full. What happens if we encounter a situation in which more than one of these rules is applicable, but there are contrasts on the prediction?
This is where sorting our rules plays a determinant role: by sorting the rules, we guarantee mutual exclusion to our model.

If we want to perform sorting, we must first decide how to rank our rules. We have already anticipated something in the subchapter about rules extraction, but here we will give the full explanation.

We have come to the conclusion that the first parameter to be consider to rank rules is their confidence. Confidence measures the probability to have B in a transaction containing A.
If we have a high confidence for the rule "A implies B", then we have high probability that if the event A happened, in the future B will happen.

The second parameter used for sorting is lift. Lift gives a measure of the significance of an association rule. The higher the lift, the stronger the bound between consequent and antecedent. It naturally came from this to decide based on this parameter which rule to apply in case they have the same confidence.

The last parameter is what solves another subtle problem. To better understand this, let us give an example. A thing I noticed while manually skimming the mined rules, was that there were situations like this:

```
(1,0,stazione_vuota) ==> (2,1,stazione_piena)
(1,0,stazione_vuota),(4,0,stazione_vuota) ==>
(2,1,stazione_piena)
```

Assuming that these rules have the same confidence and lift, it is clear that the second one gives us something unnecessary. If the first rule already tells us that if the station 1 is empty, after one time unit the station 2 will be full, it is irrelevant the information about station 4 given by the second rule.
For this reason, the third and last parameter for sorting will be the length of the rule. In this way if we have to decide among rules with the same confidence and lift, we will choose the one that performs the prediction using as little information as possible.

We have finally decided how to rank rules, which will be fundamental to perform the best predictions possible.
The way I implemented this is by introducing a class that contains all this information. The best way to explain it is to first show its code:

```
Set of Strings antecedent;
double conf;
double lift;

int compareTo(AntecedentAndStats o) {

     //This method is required to give an order to the
     //rules;

     if(lift>lift of o)
          return -1;
     if(lift<lift of o)
          return 1;
     if(lift==lift of o) {
          if(conf>conf of o)
               return -1;
          if(conf<conf of o)
               return 1;
          if(conf==conf of o) {
               if(size of antecedent<size of antecedent
of o)
                    return -1;
               if(size of antecedent>size of antecedent
of o)                    return 1;
               else return 0;
          }
     }
     return 0;
 }

 String toString() {
     String ret = antecedent + " " + conf + " " + lift;
     return ret;
 }
```

The three pieces of information require to perform ranking are stored in the private attributes of the class: antecedent (which is the set of records of the antecedent, seen as strings), and two doubles which are confidence and lift.
Of course we have the constructor class and the getter methods. I also decided to redefine the toString method for debugging purposes.
The most important method, thought, is the comparator. I had to override the compareTo method so that I could customize the sorting of my rules. Thanks to the introduction of those 3 parameters, and thanks to the format I gave to rules (the one seen in 2.4.1), that contains the 3 things I need in an easy-to-read way, writing this comparator was straightforward.

The first thing I check is which rule has the greater confidence, and return the proper value. If they have the same confidence, I do the same check but based on lift. If also the lift is the same, I check which rule has a smaller Set of records in the antecedent (which means I check which rule has the shorter antecedent). In case also this parameter is the same, I decided to return 0, so that in sorting the two rules will have random order decided by the machine at runtime.

Eventually, we can use this new Data Structure to finally sort the extracted rules:

```
//Now I want a PairRDD like this <AntecedentAndStats,
String>
Map collection of rules to collection of tuples
(AntecedentAndStats, String);

Sort collection of tuples by rules rank;
```

Of course we have first to map the rules obtained by the extraction phase, so that we can perform the sortByKey. Since we are sorting by key we need AntecedentAndStats to be the key, while the associated value will be the actual rule.
We use a mapToPair passing as argument the ToPairs class, whose pseudo-code is:

```
ToPairs {
 ForEach rule:

      Create empty set of antecedent;
      Split the rule and store the antecedent;
      Split the antecedent in its events;
      Foreach event {
           Add event to set of antecedent;
      }
      Store confidence and lift of the rule;
```

```
      Create AntecedentAndStats using the stored data;
      Create a tuple with AntecedentAndStats as key and
consequence as value;
      return tuple;
EndForeach;
 }
```

The method is simple: the first thing it does is to get the antecedent of the rule, split the records that compose it, and put them in a HashSet.
It then splits what remained of the rule, isolating the confidence and the lift and storing them into two double variables.
Eventually, it creates the AntecedentAndStats structure by means of its constructor, and returns a Tuple2 whose first element is this structure, while the second one is the consequent of the rule. This will help us a lot in analysing the predictions.

Having done this operations allows us to call the sortByKey method on the obtained collection of pairs, to finally get the sorted collection of rules. We do not have to tell Spark to sort them in decreasing order, because our implementation of compareTo takes care of it.

2.7.1.4 PREPARATION: Collecting the formatted rules

To collect the results of test, we will use a method that requires the set of sorted rules. We cannot pass an RDD since it is not an actual collection of items. For this reason, before proceeding with prediction and test, we will fill a TreeMap with our sorted rules.
This is a trivial task done by the following line of code:

```
Call collector method on the collection;
Store the Sorted Map of rules locally;
```

Of course, since it is a sorted map, I decided to use a TreeMap instead of a HashMap, so we can keep the order of the RDD.

2.7.1.5 PREPARATION: Create windows for testing

This is the last step before actually performing the predictions and testing their quality. Just like we created windows to extract the rules, we now need some testing field. What we decided to do is to take all the events that are stored in register.csv, related to the days in the range between the 15^{th} and the 30^{th}.

Since we do this, we also had to modify the part about rule extraction (which will be discussed in more detail in the proper subchapter, together with all modification I have made). We took the decision to mine rules taking in account only events from the 1st day to the 15th.
To sum up, the first two weeks will be used to train our predictor, by extracting rules from their events, while the third and the fourth weeks will be used to test the rules.

To create our testing windows, I recycled the code that I used to create the training windows, so I will not show it again here. The only thing that changes from what you can see in paragraph 2.3.1 is the addition of the following line of code:

```
//Now I need all the windows of the second fifteen days.
These will be used to test the goodness of the extracted
rules.

Apply filter to the events of the file;
Store a collection of only interesting events;
```

This filter keeps only the records whose date is between the 15th day and the 30th. The argument we pass to the FilterFifteen class constructor is the oldest date present in the register.csv file in seconds, plus an amount of seconds that elapses in 15 days.

The FilterFifteen class, in fact, keeps only those records that are not more than 15 days distant from the user-defined value.

```
FilterFiften {
 Foreach record:
      Split the record;
      Parse the timestamp;
      Convert to Timestampinsecs;
      Read the minDay decided by the user;
      if(Timestampinsecs < timestamp of minth day)
           return false;
      if(Timestampinsecs >= timestamp of (min+15)th day))
           return false;
      return true;
 Endforeach;
}
```

This code works for every set of 15 days we would like to filter, and in fact we will use it also in the rules extraction to take only the first two weeks.

2.7.1.6 TEST: Initialize confusion matrix

We are ready to test our rules and the quality of our predictor. In order to do so, we need a proper data structure, to store all the information about the process.
I decided to use the classic confusion matrix, which is a fast and simple way to test our association rules. This structure is a specific table layout that allows visualization of the performance of an algorithm, typically a supervised learning one.
In such a matrix we cross the data about predicted classes and actual classes, by putting the first in the different columns and the second in the different rows.
It is called confusion matrix because it gives an immediate visualizations of what the algorithm is confusing (wrong labels).

In our case, we have 3 labels: empty, normal, full, and for this reason we will use a 3x3 confusion matrix. The layout of the table will be like this

Predicted empty and was empty	Predicted normal but was empty	Predicted full but was empty
Predicted empty but was normal	Predicted normal and was normal	Predicted full but was normal
Predicted empty but was full	Predicted normal but was full	Predicted full and was full

Table 2.3 Confusion Matrix

The strength of this table is that it allows us to easily compute the precision, recall and F1 score of each class.
We can see that the cells on the main diagonal are the ones in which we want the highest numbers, since they represent the correct predictions.

Before starting the predictions and the testing, then, we initialize this data structure:

```
int[][] confusionMatrix = new int[3][3]; //This is the
variable to store the confusion matrix. It is also the
zero-value of the aggregate.
```

2.7.1.7 TEST: Perform predictions and test quality

We have come to the core part of the testing. We have the TreeMap containing sorted and formatted rules only about the chosen station, and an empty confusion matrix that will be the Zero Value of our aggregate operation, and we are ready to scan the testing windows,

applying the best rule for each one and confronting the prediction with the actual event, putting the result in the proper cell of the confusion matrix.

We use the Spark action "aggregate". This, in fact, allows us to compress the information from an RDD of any type into a Java variable of any type. In this case, we will go from JavaPairRDD<AntecedentAndStats, String> to int[3][3], a.k.a. our confusion matrix.

```
Apply Aggregate to the events collection;
Store the final value of the confusion matrix;
```

To call the aggregate() in Spark, we need 3 things: the zero value, the sequential operator and the combining operator.

We already said the zero value is the empty confusion matrix.

The sequential operator is the most interesting part. It contains all the logic that is behind this testing process: it scans the testing windows one by one, and for each one it looks for the most fitting rule among the sorted set. When it finds it, it checks the prediction and the actual event, and fills the relative cell of the confusion matrix.
If there is not a rule that fits the window, the prediction is "normal_state".

This operator requires us to pass two arguments to it: the TreeMap with the sorted rules, and the ID of the station we are analysing. This is why we had to collect the rules in a Java variable, otherwise we would not have been able to perform the scan for each window.
The station, on the other hand, is needed for some String manipulations we will do to understand if the prediction is correct.

```
Sequential Operator {
 Foreach window:

      //First, I create a set containing all events in
the window.
      Split the window;
      Collect the events in a Set;


      //Now we check what is the best rule that fits the
event.
      Foreach Rule in the sorted map {
           Read antecedent of the rule;
           if(antecedent is contained by set of events of
the window){
```

```
                Read prediction;
                Store Type of prediction;
                if(prediction is in the window){
                     if(typeOfPrediction is empty){
                          arg0[0][0]++;
                          return arg0;
                     }
                     if(typeOfPrediction is full){
                          arg0[2][2]++;
                          return arg0;
                     }
                }
                else {
                     Invert prediction;

                     if(inverted prediction is in the
window) {
                          if(typeOfEvent is empty) {
                               arg0[0][2]++;
                               return arg0;
                          }
                          if(typeOfEvent is full) {
                               arg0[2][0]++;
                               return arg0;
                          }
                     }
                     else {
                          if(typeOfPrediction is empty){
                               arg0[1][0]++;
                               return arg0;
                          }
                          if(typeOfPrediction is full) {
                               arg0[1][2]++;
                               return arg0;
                          }
                     }
                }
           }
      }
      //If I did not find any rule matching, it means
that the prediction is "normal state".
      String empty = "("+station+",1,stazione_vuota)";
      String full = "("+station+",1,stazione_piena)";
```

```
        if(events contains full){
             arg0[2][1]++;
             return arg0;
        }
        if(events contains empty)) {
             arg0[0][1]++;
             return arg0;
        }
        else
             arg0[1][1]++;
        return arg0;
    }
```

As we can see, the sequential operator is given a temporary confusion matrix and a String (representing one of the testing windows). It provides in output the updated confusion matrix, after having analysed the window. In other words, it will increment the proper cell, based on what the prediction and the actual event for that window are.

We need to point out a simple thing. We have already seen in the preparation part that we shifted all the rules whose consequent was relative to timestamp=1 of one time unit. In the explanation that follows, why we did this will be clearer. In the Sequential operator, in fact, we are assuming that predictions and actual events are always relative to timestamp=2.
When I first performed the tests, I did not shifted such rules, and for this reason I was always getting a that the actual event was "normal_station". Introducing this shift solved the problem and lead me to correct results.

For implementation of the Sequential operator, the first thing it does is to create a Set of events from the window, by splitting the String and adding each substring (an event) into a HashSet.
Then it loops to find the most fitting rule. Since rules are already sorted by decreasing rank, we can perform this with a time complexity O(n). We simply scan down the key set of the map of rules, and the first rule that matches the analysed window is considered the best one. In order for a rule to match the window, the latter needs to contain ALL the event in the antecedent of the former.
At this point it extracts the prediction from the consequent of the selected rule, noting what type of prediction it is (empty or full). If no rule matches the window, than we assume that the prediction is "normal".
We then have to understand what is the actual event in the window, so we can increment the proper cell of the matrix. This can be tricky.
We first check if the prediction is present in the window. If it is, we will increment the cell on the diagonal corresponding to that type of prediction. In fact, cells on the diagonal refer to correct predictions, as we have already seen. In case the prediction is "normal", then we say it was successful if there are no events about that station at time 2.

If the prediction is not present, we create the inverse of the prediction, by calling the following invertPrediction method:

```
String invertPrediction(String prediction) {
      Split prediction;
      if(event is "empty")
           return "full";
      if(split[2] is "full")
           return "empty";
      return null;
 }
```

This piece of code is simple: if the prediction is "empty" it replaces it with "full" and vice versa.
Now we check if the "inverse event" is present in the window. If it is, we put it in the corresponding cell, otherwise, we assume that the actual event is "normal" and we increment the relative cell.
The procedure for "prediction = normal" is a little different, since we do not invert the prediction (it would not make sense), but we just check which of the two possibilities is present. If none of them is, we increment the cell corresponding to correct normal prediction.

The third element required by the aggregate method is the combining operator. In this case, the combination consists in merging two temporary confusion matrixes (generated by the sequential operator or by previous combinations). The code to do this is trivial:

```
Combining Operator {

      int[][] res = new int[3][3];
      for(int i = 0; i<3;i++)
           for(int j=0;j<3;j++)
                res[i][j]=arg0[i][j]+arg1[i][j];
      return res;
 }
```

As we can see, the call method takes the two confusion matrixes as arguments, and perform the sum cell by cell, returning the resulting confusion matrix.

We have finally got our results, ready to be stored and checked, to see how our predictor is behaving.

2.7.1.8 TEST: Format and store results

Before storing the results in the HDFS we need to find a proper formatting. During my experiments I used two different formats.
At first, I was using something like this:

```
Predicted empty and was empty: x
Predicted empty but was normal: y
Predicted empty but was full: z
Predicted normal but was empty: v
Predicted normal and was normal: w
Predicted normal but was full: k
Predicted full but was empty: i
Predicted full but was normal: j
Predicted full and was full: m
```

The pros about this choice is that results are easy to read for a human user and creating manually a .csv file is trivial.
Still, it was annoying to manually write each line of this file. When I started extracting statistics for all the 284 stations, I decided to pass to a new format:

```
Id    x    y    z    v    w    k    i    j    m
```

Of course, this is less readable, but it was way faster to just copy and paste each line of my statistics file, and so I still used this.

```
String result = "predicted empty and was empty:
"+confusionMatrix[0][0]+
                "\npredicted empty but was normal:
"+(confusionMatrix[1][0]+confusionMatrix[2][0])+
                "\npredicted normal but was empty:
"+(confusionMatrix[0][1]+confusionMatrix[0][2])+
                "\npredicted normal and was normal:
"+(confusionMatrix[1][1]+confusionMatrix[2][2]+confusionM
atrix[1][2]+confusionMatrix[2][1]);



      Print result;
```

In order to save something in the HDFS we have to make it an RDD, so I used a dummy RDD, filling its only record with the string containing the information about my confusion matrix, using the method parallelize.

At this point, we use saveAsTextFile, and save the statistics in a file whose name contains both the threshold used, and the station chosen by the user. This was very helpful to understand what statistics were relative to what stations.

2.8 Conclusions

At this point, we have described the whole process to extract rules, and to later test the quality of the classifier based on them. In the next chapter I am going to give an exhaustive explanation of all the experiments I have conducted during the work on this thesis, using the classifier I have just described.

Chapter 3
Experimental results

3.1 Objectives

After having given an exhaustive description of all the process to extract and test rules, I am going to discuss the experimental results.
My objective was to improve the quality of the classifiers based on classic methods, by introducing some considerations on time and space relationships among events. To do so, I had to stress in different ways the predictor described in the previous chapter.
Rule mining, in fact was only a small part of the work. The most challenging phase was performing tests and tuning parameters to adjust the results and find the optimal trade-off between computational performances and quality of the predictions.
In this chapter I am going to show in detail all the tests I did on my classifier, I am going to discuss the results of each testing phase and eventually offer proper explanation of what I had to change, redefine or even add to the whole project and why.

3.2 Work material

The first thing I am going to examine is the material I used for my purposes. This will make every other step much more clearer and easy to understand.
The case of study, as we have already seen, is about the states of the bicycle stations in the city of Barcelona. We are going to analyse their critical states and try to understand if some patterns are frequent. For this reason, I was given two .csv files: *registers.csv* and *stations.csv*.

3.2.1 The register file

The core part of the information I needed to train and test my classifier is stored in this very big file (of more than 700 MB). This is a .csv file, whose separator is a tabulation character, storing in each line the information about the state of a station at a given time.

The file contains about 27 million of lines, and it is the collection of what sensors placed on 284 different bike stations of Barcelona took over in 4 months and a half (from 15-05-2008 to 30-09-2008), by taking a measurement every 2 minutes. This alone should give the idea of the amount of data we are going to process.

The format of each line is the following:

```
id    time&date free_slots    used_slots
```

and it gives us the number of slots already occupied by a bicycle and of the ones still free at a given time for the station with the given id.

This file will be processed both for training and testing the classifier. In particular, the information about the first 15 days will be used to extract the rules used by the classifier, while the second 15 days will be our testing data set.
It is important to point out again that the actual format we will use for the interesting events is the following:

```
id    timestamp      type_of_critical_event
```

obtained after the transformations reported in the previous chapter.

3.2.2 The stations file

This file contains some interesting subsidiary material. It is a much smaller .csv (still with a tabulation as separator), of 175 KB, storing geographical information about bike stations of different European cities. In particular, each line has the following format:

```
id    latitude  longitude      name_of_station
```

The file contains more than 3000 records (corresponding to the same amount of different bike stations), but we are interested only in the first 284, which are the ones relative to Barcelona stations.

This file will be used in the last experiments we will see, to retrieve information about relative distance of the various stations. In fact, while at first we are interested only in the time relationships among events, at the end of the work I started taking in consideration also the constraint of space distance. This will come in handy to filter out something that can be seen as not very useful, and improve performances without losing quality.

3.3 Usage of the classification testing tool

In Chapter 2 I reported the whole process that leads from the records in the register file, to the extraction of rules, converging eventually into classification testing.
Here I am going to describe, instead, **how** to use the tool, in order to obtain the results.

I would like to introduce all the different applications I used in this brief subchapter, and then describe them more in detail in the parts about experiments where each one of them were used.

3.3.1 ExtractRules

This was the first piece of code implemented during my work, and it has only the first part, where the register file is scanned and rules are extracted, properly pruned and stored in the HDFS. As we will see later, at first this code took in account **all** the records about the stations, and for this reason it was very heavy. After the introduction of the sampling time interval, the weight on the machine significantly decreased, and I was able to use smaller values for the minimum support.

To use this tool we need to call it adding some parameters:

InputFile: it is a string containing the path of the file with the dataset to analyse. In this particular case of study, the file will always be *registers.csv*.

Interval: this integer value is the sampling interval expressed in seconds, that can be defined as the dimension of a time unit in our rules. For example, a value for the Interval of 3600 means that only records distant a multiple of 1 hour from the starting instant (15-05-2008 12:00:00) are kept, while the rest is discarded;

Window: this is another integer, and it is the dimension of the temporal window, a crucial parameter to extract rules. The dimension of the window defines how many events can be considered related among them. For example, if we have a window of 4 time units, it means that we are considering events distant at most 4 time units correlated. Choosing a too high value for this parameter makes the complexity of the extraction process explode.

MinSupport: it is a double that defines the minimum support for the rule mining algorithm. FP Growth will keep only those rules that refers to transactions that pass this filter. Ideally, we would like to keep this value as low as possible, but this was one of the hardest obstacles. If the chosen value is too small, the complexity explodes, but if it is too big, we get only rules about a few stations.

MinConfidence: it is a double that defines the minimum confidence for the rule mining algorithm. Only relationships with a frequency higher than this value will not be discarded. Tuning this parameter is less challenging than tuning the minimum support, but in this case we want to keep this value somewhat high (not too much or we will filter every rule).

OutputFolderRules: it is simply a string representing the output folder of the HDFS in which the application will store the extracted rules.

Station: this integer parameter allows the user to define the station id which to perform the process for. Only rules with this particular station in the consequent will be extracted, and the testing results will be relative only to predictions about the state of this station.

Threshold: this parameter is an integer value that defines when a station can be considered empty or full. This threshold refers to both cases, so if we choose 3, only stations with **less** than 3 free slots will be marked as empty, while only stations with **less** than 3 used slots will be considered full.

3.3.2 TestRules

This application is the skeleton to all the following ones. After having manually checked the type and quality of rules extracted by the previous application, I added all the functionalities (reported in the previous chapter) to perform testing of the classifier.
So, just like ExtractRules, this application starts from the register.csv file, but instead of producing in output a file containing the mined rules, it prints out, in a proper format, the results of the classifier; in other words, it prints the confusion matrix, that we will use for our statistics.
It still perform the ExtractRules code, of course, but the mined rules are stored in a proper data structure and immediately used, without printing them out (which can be very heavy, due to the number of rules).

The arguments required by this application are exactly the same of ExtractRules.

3.3.3 DumbExtractor

This is a logically simple application, but it was crucial in different parts of my work. It is called “dumb” extractor because it is actually a copy of TestRules, but the tested rules are

not the ones extracted by means of my solution. Instead, it uses a "dumb" system to predict future events: the prediction is always the current state.
For example, if I have

```
1     1     stazione_piena
```

This classifier will predict that:

```
1     2     stazione_piena
```

Of course this approach is way faster than rule mining, but the point was to get a baseline to make some comparisons, and to try to understand if it was worth to perform rule mining.

Some details of the implementation of this will be discussed in the proper subchapter.

3.3.4 TestRulesWithDistance

This last application does all the things of TestRules, but with the introduction of the constraint about distance between stations.
I will go more in depth about this in the dedicated subchapter, but, as a general idea, I would like to improve performances (and actually tackle other issues, too) with little to none performance loss. If I accept the idea that two distant station have no influence on each other, then introducing this new parameter (and the relative code that takes care of it) can be useful.

The application has the same parameters of TestRules, but one is added at the end:

MaxDistance: a double that expresses, in kilometres, the radius of the circumference I want to consider in my analysis. For example, choosing 1.5 as Maximum Distance is like opening a compass of 1.5 km, pointing it in the analysed station, tracing a circle, and discarding all stations not included in that circle.

3.3.5 Script

At last, I also wrote a script that allows to launch the previous applications once for each and every one of the 284 stations automatically. In fact, we saw that when we use them, we have to specify the stations for which to perform the actions. If we want to have an execution for every station, we launch this script. It is important to notice that the script

sleeps 30 seconds between each execution of the application to prevent overloading the cluster.

3.4 Experiments

Conducting experiments on the given data set was the main part of my work. In fact, once I managed to write code to extract rules and to test them, I started using it on *registers.csv* to understand if it was worth using this new approach or if it would just increase computational complexity without leading to better results.
A very important thing to note, indeed, is that my approach is surely less efficient than the classic ones, and it is advisable to use it only if classic methods do not bring to satisfying results.

In this part I am going down with details about all experiments I did on my data set. Each experiment required some modifications to the code and some tuning to the parameters, which I will discuss in details in every subchapter, so that everything is as clear as possible.

I am going to accompany the discussion with tables, data and graphs, to show in the best way possible why I believe a solution is good or bad.

3.4.1 Statistics about windows and rules

When I first started conducting my analysis, I met a lot of issues due to not having a clue on how to dimension parameters. During my work I came to understand that lots of parameters are firmly connected among them, and that dimensioning them properly is the key to obtain significant results in an efficient way.

In particular, in the first steps I had problems with the dimension of the windows and with the minimum and maximum support and confidence. I was using ExtractRules to check what types of rules my classifier would end up using, but more often than not the Extraction did not terminate correctly it execution or would give me empty files.

For this reason I decided to collect some statistics about how different parameters influenced a number of crucial things, about transactions and rules.

Here I will report the results obtained both with tables and with graphs.. The window was pretty big, with a dimension of 6, and keeping this value fixed, I modulated the interval, the confidence and the support.

3.4.1.1 Results on transactions

Interval	Count	Avg	Max	Min
10 min	17939	776	1141	91
30 min	5982	764	1111	85
40 min	4459	760	1189	79
1 hr	2988	749	1074	82
2 hr	1469	726	1029	97

Table 3.1 Statistics on transactions

Table 3.1 shows how some parameters about the transactions change at the variation of the sampling interval, keeping the window fixed at 6.
In particular, Count is the number of transactions created starting from the register.csv file. Of course, if we increase the sampling interval, this values decreases, since we are taking into account much less events. It is interesting to note that Interval and Count have a linear relationship: if I double the sampling interval, the number of transactions halves down.
The other three parameters refer to the length of the transactions (a.k.a. the number of elements in a transaction). We can see how the average length decreases when we increase the sampling interval. The same happens with the Max length of transactions: of course if we decrease the interval, we have smaller transactions. The Min length, instead, looks like it is not correlated: this parameter is not affected by the interval because it is completely random to take a smaller transaction. It may happen, in fact, that using a sampling interval of 60 minutes brings to take into account a very small transaction, not taken into account if we, instead, use a 25 minutes interval.
In the end, we can see how increasing the Interval allows us to get less transactions, and smaller ones.
At the beginning I was not sampling my input data file, and I was getting so many transactions and so big, that it was impossible to extract rules due to complexity.
I, then, decided to check with a simple code what were the best values for the sampling interval parameters.

As we can see, the Count, which is the number of generated transaction, is inversely proportional to the interval. If I choose an interval three times bigger, I get a number of transaction three times smaller.

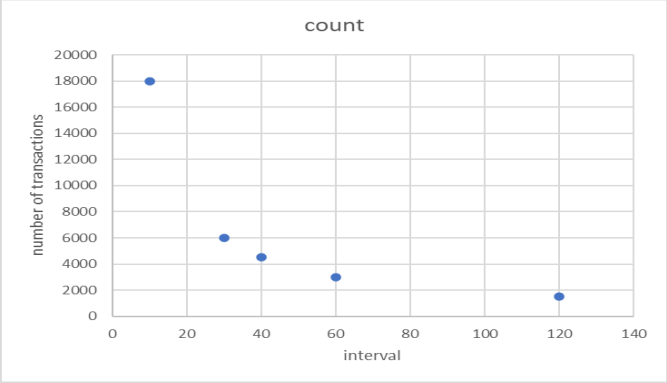

Figure 3.1 Number of rules and interval

An interesting thing to point out is that the maximum and minimum length of transactions are not influenced by the interval, but they have “random” fluctuations for which we are not able to find an explanation based on the interval. The average length, however, goes down as expected, but not of a relevant value. This means that of course by incrementing the interval, the complexity of the extraction process decreases, but it is not because of decreased length of the transactions, but only due to having less transactions to process.

3.4.1.2 Results on rules

Interval = 30 minutes

Min support	Number of rules
0,45	53519
0,5	6916
0,55	996
0,6	170
0,7	3
0,8	0

Table 3.2

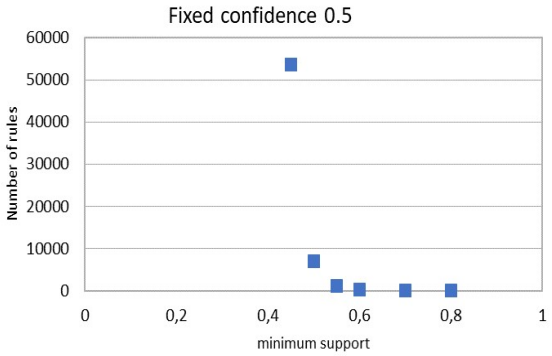

Figure 3.x

Min confidence	Number of rules
0,5	53519
0,55	53519
0,6	53439
0,7	51905
0,8	45791
0,9	29922
0,95	2501

Table 3.3

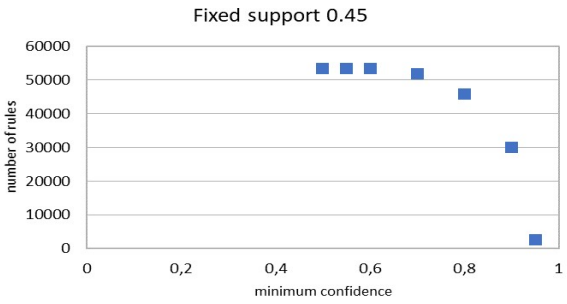

Figure 3.3

Interval = 60 minutes

Min support	Number of rules
0,45	8604
0,5	1577
0,55	280
0,6	51
0,7	1
0,8	0

Table 3.4

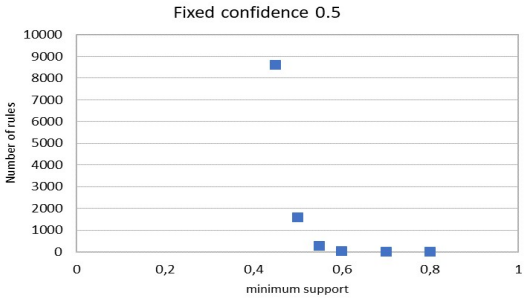

Figure 3.4

Min confidence	Number of rules
0,5	8604
0,55	8604
0,6	8538
0,7	7316
0,8	4809
0,9	877
0,95	4

Table 3.5

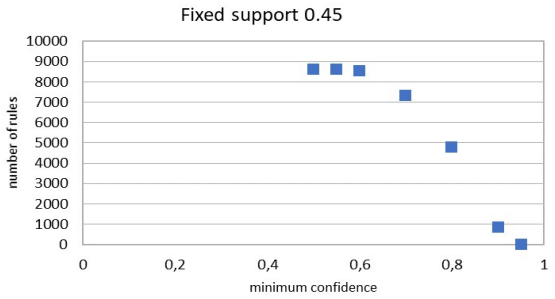

Figure 3.5

Interval = 120 minutes

Min support	Number of rules
0,45	2388
0,5	558
0,55	98
0,6	15
0,7	0
0,8	0

Table 3.6

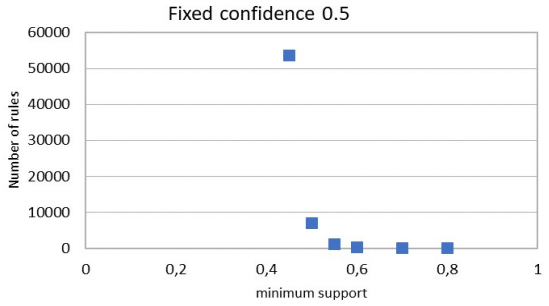

Figure 3.6

Min confidence	Number of rules
0,5	2388
0,55	2388
0,6	2334
0,7	1315
0,8	503
0,9	5
0,95	0

Table 3.7

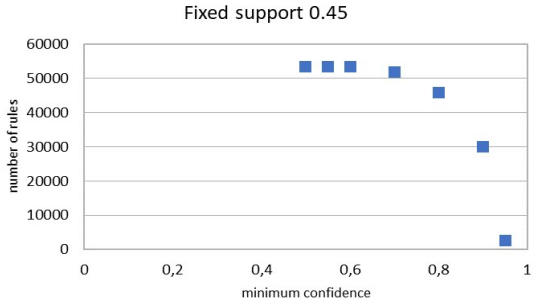

Figure 3.7

The above tables and graphs show how the number of rules varies accordingly to the different intervals, supports and confidences. In this specific situation we are considering all the stations in the file (even those for which we will not be able to extract any rule). The limitation about what stations to consider in the analysis will be introduced later.
As interval I chose 30, 60 and 120 minutes because are the ones that give good results in a reasonable amount of time. Furthermore, I consider making previsions about what will happen in 30, 60 or 120 minutes much more interesting than predicting things that will happen after 2, 5 or 10 minutes.

From the tables it is clear how increasing the sampling interval drastically decreases the number of extracted rules, and, of course complexity of the process.

From these considerations stems that the support and the confidence influence the amount of information that the classifier will be able to use, to label data.
These graphs show immediately that keeping the minimum support as low as possible gives the best results, because it extracts many more rules (because more stations make it past the filter, and manage to get rules about them).
On the other hand, we can keep the confidence high (about 80-90%) since we will be able to extract a sufficient amount of rules, and with this high value we will only get the most meaningful ones. Remember, in fact, that when we perform the tests, the rules are sorted by confidence, and keeping those rules with a low confidence will just waste memory and resources.

From these results, I took the decision to proceed using the intervals of 30, 60 and 120 minutes, while trying to go as down as possible with the support, and keeping a high value of confidence.
Lastly, I came to the conclusion that a window of length 6 is not adequate. It is too long, and the average length of the generated transactions is so big that we end up extracting way too many rules. For this reason, I reduced the window dimension to 3, in the following experiments (besides when I specify a different dimension).

3.4.2 Initial tests

After these considerations on the rules and on how to tune parameters to perform rule extraction, I started performing actual tests.
In the beginning, I had to face different issues, that prevented me to obtain usable results.

First, there is a very high heterogeneity among the different stations. There are about 10-15 stations that are most of the time in an empty state. Analysing their position, I could see that they were placed in not very popular zones.
This causes a big problem, because when performing the filter to take only records about critical situations, more than 85% of remaining records are about them. This means that I will extract mostly only rules about them. The solution would be to lower the support as much as I can, so I can take into account also other station, but doing so had proven to be impossible multiple times: the complexity explodes, and the rule mining algorithm fails.

Another problem is represented by the extreme heterogeneity of the classes. This is due to the definition of the classes themselves: it is way more easy to end up in the empty critical state (which in this phase consist in having less than 3 used slots) than in the full critical

state (less than 3 free slots). Furthermore, the first problem about bike stations in not popular places has the effect to increment this heterogeneity even more.
This translated in the impossibility to get rules about full critical state. I will tackle this issue in a future experiment.

3.4.2.1 Results about quality of predictions

As already said, the stations in unpopular places prevailed, and in this first phase I was able to extract statistics only about them. It is still very interesting to see how the classifier performed, to have at least a general idea about this new approach to classification, and to be sure it is worth going on.

I decided to implement a simple piece of code that printed in output the ids of only the stations that have at least one rule extracted. In this way I avoid wasting time performing tests for stations with 0 rules. To see what happened with these, though, I decided to perform a couple of tries with them. The result was expectable: since there are no rules about them, the prediction is **always** "normal state". The good side of this is that this prediction is correct more often than not. In fact, stations for which no rules were extracted, were in the normal state much more times than they were full or empty. This lead to satisfying results.

The most interesting part, however, is about the stations for which the classifier had rules to apply. In this first phase I managed to extract statistics for 48 stations (the least popular ones).
I will show here only data about some of them (station 213 which was the best, and 211 which was the worst), plus an average on all the statistics.

	Actual event: empty	Actual event: normal	Actual event: full
Prediction: empty	228	71	0
Prediction: normal	28	9	0
Prediction: full	0	0	0

Table 3.8 Confusion Matrix of station 213

	Actual event: empty	Actual event: normal	Actual event: full
Prediction: empty	87	174	3
Prediction: normal	23	49	0
Prediction: full	0	0	0

Table 3.9 Confusion Matrix of station 211

From the table 3.8 we see a very good thing: the largest values are on the diagonal of the confusion matrix. This means that the classifier is already doing a good job, with the exception of the full class for which we do not have any rule. The vast majority of stations behave like this one, since they are often in the empty state (so prediction based on rules becomes accurate) and in the rare cases where no rules apply, the prediction of "normal state" is actually correct.

Table 3.9, on the other hand, represents the worst case I got with this classifier. There are few stations that behave like this, but it is still worth noting.

From the confusion matrixes of the 48 stations analysed, we were able to extract some noteworthy statistics:

P(E)	R(E)	F1(E)	P(N)	R(N)	F1(N)	P(F)	R(F)	F1(F)	Acc
0,66	0,83	0,72	0,43	0,26	0,30	N/A	N/A	N/A	0,62

Table 3.10 Statistics of the predictions

The results of table 3.10 were obtained using a window of size 3, interval of 1 hour, support of 0,45 and confidence of 0,8. I conducted analysis taking into account only this 48 stations. I computed Precision, Recall and F1-score of the three classes for each of these stations, and then I computed the average for each of these statistics.

With this first approach, even if with plenty of issues, we managed to obtained an accuracy of over 60%. I considered this a good result to begin with, since it was obtained with little to no tuning of the parameters. However, as we already said, the dataset and the class distribution were so heterogeneous that Accuracy is not very indicative of the quality of the classifier.

Looking at the more interesting parameters of precision, recall and f1-score of the classes, we notice that:

Empty class: the classifier does a very good job in labelling events of this class. 66% percent of empty labels are actually put on empty events (precision=66%), and over 8 empty events out of 10 are classified correctly (recall=83%). This gives us a very good f1-score for this class of 72%, much higher than the accuracy. This is due to the fact that the stations considered here are the ones usually empty.

Normal class: the results concerning this class are way less thrilling. A precision of 43% tells us that over 1 out 2 events labelled as in normal state, are actually empty or full. What

is even more lacklustre is that only 1 normal state out of 4 was correctly labelled. This brings to an f1-score of 30%, half the accuracy of the classifier. Again, these stations are so often in the empty state, that even when we have no rules about them, there's a high chance they're empty.
By the way, from the table above we notice immediately that these statistics are worsened by those unpopular stations we already mentioned. For example, stations like 203, 213 and 238 are nearly always empty. There is a high chance, then, that even when we predict that they will be normal, because there are no rules about that particular situation, they will end up being empty. However, for stations that are not in that limit case, we obtain much better results, with F1-scores that go from 40% to 60%.

Full class: we did not manage to extract information about this class. The heterogeneity of classes is too high, and only in very rare cases stations are full. This brings not to extract rules about the full state, and as we see in the table, even during the testing phase, full states are much less frequent than normal and empty.

3.4.3 Binary predictor and threshold tuning

The next step in my work stemmed from the considerations we made about the full class. Events in this class are so rare that it was not even possible to extract information about it. We decided, then, that dropping one type of label, and making our classifier a binary predictor could be more than acceptable, simplifying the process and leading to more significant results.

What I did is to merge the normal class and the full class into the new "Not-empty" class. If a station is normal or full, now we simply consider it not empty.
Thanks to this modification, I had more freedom to try and decrease the support, and tune the threshold that defines an empty station, to see if I can get better results.

The full class is, for now, left behind. We will consider it again later, but it is actually not very interesting, since it is so difficult for a station to end in such a state (as we can see in table above). We could try to lower the threshold to be considered full, but then what would be the point of this analysis?

3.4.3.1 What did I change?

Making this classifier binary was actually very simple. I did not change the code at all, the only thing modified is how the confusion matrix is printed in output. In fact, I still fill a 3x3 matrix, but then I englobe some cells together. For example, the cell with the number of

"predicted full but was empty" and "predicted normal but was empty" become one single cell called "predicted not empty but was empty".

About the change of threshold: this is a user-defined variable, so playing with different values was straightforward.

3.4.3.2 Results of the binary classifier with threshold tuning

We get some interesting results from this experiment.
First we see how not influent was the removal of the full class. Englobing it with the normal class lead to same or better results, with a much lower computational time.
Secondly, we immediately see that decreasing the threshold to be considered empty worsens the prediction quality about the empty class, but increases the one about the not empty class. The graph below (figure 3.8) shows this evidence. These results were obtained using the same parameters of the previous experiment. As seen for the previous experiment, these average values are obtained taking into account only the 48 stations for which we are able to extract at least one rule.

Threshold	P(e)	R(e)	F1(e)	P(ne)	R(ne)	F1(ne)
1	49%	61%	52%	70%	58%	61%
2	60%	78%	66%	53%	34%	38%
3	66%	83%	72%	45%	27%	31%

Table 3.11 Variation of threshold

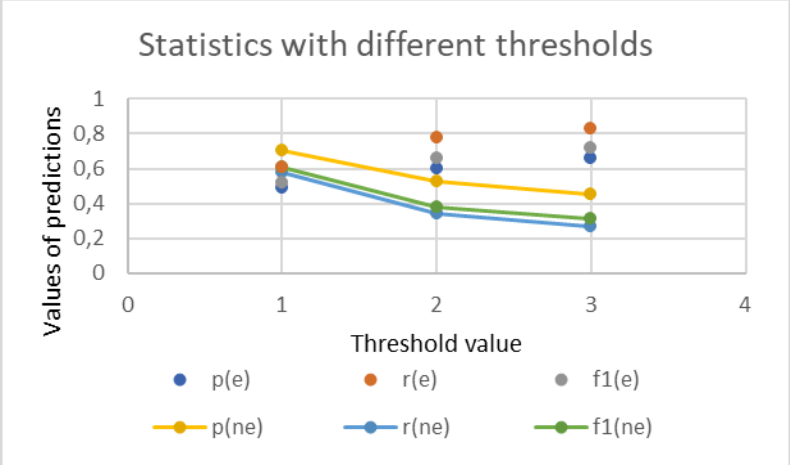

Figure 3.8 Quality of predictor at variation of threshold

This seems logic since it is more difficult to end up in the empty class, and so the predictions about that.

However, it is evident that decreasing the threshold to 2 and especially to 1 increased a lot the performances of the classifier. With the latter value, in fact, we have a f1-score for the empty class of 52%, f1-score for the not-empty class of 61% (huge improvement!) and an accuracy of 60%.

3.4.4 Decreasing the window dimension

Until now we have described experiments carried on using a temporal window of dimension 3. The sampling interval has been of 1 hour for all our experiments, which means we were trying to predict what would happen in 1 or 2 hours.

However, we were curious to see what would change if we reduced the window to the smallest dimension possible: 2. In this case, since we are always using a sampling interval of 1 hour, we will be able to perform classification only of events that happen exactly after a hour. It still looks interesting, so it is worth giving a chance. Furthermore, decreasing the window size causes the performances to increase since there are less data to process.

Another positive aspect of decreasing the window size, is that we may be able to decrease the minimum support, as well, and managing to perform tests on more stations.

Now, the window size is a parameter definable by the user, so, once again, I did not have to change anything in the code.

3.4.4.1 Results about the impact of the window size

The results were pretty interesting. We notice a general improvement both in performance (as we expected) and in quality. It looked like discarding events distant of two time units was a rewarding choice that let our classifier work better.

	Actual event: empty	Actual event: not empty
Prediction: empty	196	25
Prediction: not empty	0	97

Table 3.12 Confusion Matrix of station 279 (best one)

	Actual event: empty	Actual event: not empty
Prediction: empty	124	113
Prediction: not empty	0	81

Table 3.13 Confusion Matrix of station 210 (worst one)

We see from table 3.12 and 3.13 that we have a huge improvement in both the best and the worst case. Aggregate results show this even better.

Table 3.14 gives aggregated information about the testing results. This was computed performing the average on the 48 stations for which I have at least one rule.

P(E)	R(E)	F1(E)	P(NE)	R(NE)	F1(NE)	Acc
0,76	1,00	0,86	1,00	0,48	0,61	0,81

Table 3.14 Statistics with window = 2

The first thing to notice is the great improvement in accuracy. Using the same parameters with a window of size 3 gave us an accuracy of about 60%, while now we went over 80%, with an improvement of 33%.
Again, labels are not perfectly balanced, so it is better to look for other parameters.

Empty class: this class was already performing good with the three-labels approach, but now it is even better: we manage to label **all** the empty events in the correct class, and 3 out of 4 empty labels we apply are correct. This brings an excellent F1-score of 86%, much better than the previous 72%.

Not empty class: this new class englobes both the previous full class and normal class. Results are pretty satisfying, with a perfect precision score, and a recall of about 50%, considering that we're still working with stations that are nearly always in the empty state. A F1 score of 61% represents a huge improvement with respect to the solution with three classes.

3.4.4.2 Differences between the two sizes

It is interesting as well to compare the results between the approach with window size = 3 and the one with window size = 2, by showing the differences in quality.

As we can see in the table below (3.15), the average precision, recall and f1-score improved for both classes (we're comparing normal class of three-classes approach with the non-

empty class, since the full class did not give us any result). There results are the average of differences computed on the usual 48 stations. A thing to be noted is that not only the averages show the improvement, but every single station had better result with the binary approach.

ΔP(E)	ΔR(E)	ΔF1(E)	ΔP(NE)	ΔR(NE)	ΔF1(NE)
+0,11	+0,14	+0,12	+0,57	+0,27	+0,33

Table 3.15 Differences with window = 2 and window = 3

The differences in quality are more than enough to justify the simplification made to our solution, by decreasing the size of the window. In particular, we see that the Not Empty class gets a huge improvement in precision (almost 60%) and more than a 25% improvement in recall, which means an increase of 33% of the f1-score.

Nonetheless, the Empty class significantly improves too, since all of his parameters increases for more than 10%.

3.4.4.3 Differences between the two sizes with threshold tuning

After having noted that results with window size = 2 were practically always better than the ones with window size = 3, I decided to see what could happen if I changed the threshold of "empty station", like we did in a previous experiment.
It could be interesting, in fact, to see if the decreased window size is generally better for the classification problem, or if there are situations in which a larger window leads to better results.

As done before, I started stressing out my tester using different threshold (and different supports as well, the smallest possible each time), and I obtained the following interesting results.

The following results were obtained using window size = 2, confidence = 80% and sampling interval of 1 hour. Support changed accordingly with the threshold, and I used 0,45 for window = 3, 0,35 for window = 2, 0,15 for window = 1. I computed the confusion matrixes of the usual 48 stations, and then performed the averages.

Threshold	Δ P(e)	Δ R(e)	Δ F1(e)	Δ P(ne)	Δ R(ne)	Δ F1(ne)
1	0,05	0,37	0,16	0,30	0,23	0,27
2	0,06	0,22	0,13	0,47	0,15	0,24
3	0,11	0,14	0,12	0,57	0,27	0,33

Table 3.16 Difference of the two windows at variation of threshold

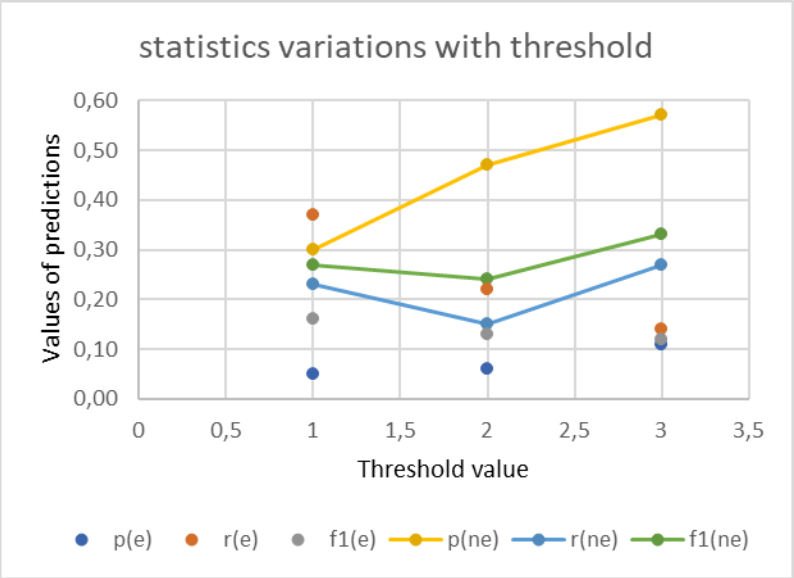

Figure 3.9

We notice immediately that the improvement is general, and so that decreasing the size of the window is probably the best thing to obtain a stronger classifier.
The greatest improvement is always in the Not Empty class, and it is intuitive, since it is likely that the normal state in a given moment is going to influence much less what happens after two hours, than what happens after only one. If a bike station, in fact, is in a normal state right now, I can imagine that the chance that this influence the normal state in one hour is way higher than the chance that it can influence the state in two hours.

Another pattern seems to emerge from these data. The precision increases if we increase the threshold while the recall decreases. This means that if we allow more events to be considered “critical”, than the chance that I label an event correctly is higher, but at the same time I tend to find less events of that class.

3.4.5 Less frequent stations

All the reasoning we have been doing until now have an important limitation. As we said, all the data collected, showed and discussed in these chapters are about those unpopular stations that appear very frequently in a critical (empty) state, because most of the time they are free.
For this reason, after the filter we use to remove uninteresting states, they overwhelm the whole file with records about them, and extracting rules about stations that have a more "normal" behaviour becomes impossible.
The problem is that these stations are less than 50, among the 284 we are analysing, and they really represent a particular case, so we may end up overestimating the performances of our classifier. Each one of these stations is present in at least 1,5% of the records (some goes over the 2% of presence), while the vast majority does not reach the 1% of presences in the critical records.

In this part I am going to describe how I faced this issue, the results I obtained and some important conclusions I was able to derive.

3.4.5.1 Procedure to get the less frequent stations

Since the beginning of this chapter I had to stress the importance of the minimum support parameter that we pass to the application. Only those transactions that appear in the file a sufficient number of times manage to pass this filter. Since we want to take into account also stations that are not that often in a critical state, we can try to decrease the support.
After many tries in this way, I had to give up. As we saw in the first graphs, the number of transactions increases exponentially if we decrease the support. This leads to an uncontrollable complexity explosion of the process, making it impossible to terminate correctly.
Even when I discussed the decreasing of the threshold (and with it, the decreasing of the support), I got a counter-intuitive outcome. Decreasing the threshold and the support made me get information about less stations than before. This, actually, has its logic. If I decrease the threshold, it means that only those records about **very** critical states are considered. This goes in favour of the often critical stations, and they overwhelm the analysis even more.

Introduction of banned stations
Decreasing the support, then, is out of consideration. I had to take a different path in order to solve this issue.
This new path consisted in the introduction of a concept: the banned stations. After having collected data about those frequent stations, I decided to modify my code, introducing a new filter that discards all the records about those stations. In this way, I was able to obtain

rules with a consequent referring to most of the less frequent stations (unfortunately, I did not manage to get at least a rule for each one of them).
There is a problem about this approach, though. If it is true that by filtering out those particular stations, rules with their IDs do not overwhelm my analysis, it is also true that we lose the possibility to have them in the antecedent of other rules. This leads to incomplete rules and, as a consequence, to possibly incorrect results.
However, after having watched carefully the data, and having reasoned on them, I decided that this "sacrifice" was actually acceptable. In fact, these stations are almost always in a critical state, and so, even in the antecedent of rules about other stations, they would always appear as empty. This may mean that the fact that these stations are empty does not have any influence on the state of other ones.

3.4.5.2 Results of classifier with less frequent stations

I was finally able to obtain results about stations that I never saw in the previous ones. I analysed the extracted rules, and I noticed that they were much more balanced, without some stations overwhelming the whole rules collection. Not all the stations have been taken into account though, because some of them are actually nearly always in a not empty state.
This is not a big flaw, since we can say that if they are usually in that state, we can simply predict that they will be in a not empty state and have a high chance to be correct.

The following results are obtained by using a window size = 2, threshold = 1, minimum support = 0,05 and minimum confidence = 0,8. I managed to obtain data about more than 90 stations. I computed the confusion matrix for each one, and then got the average.

P(e)	R(e)	F1(e)	P(ne)	R(ne)	F1(ne)
0,44	1,00	0,60	1,00	0,83	0,90

Table 3.17 Statistics of less frequent stations

It is very interesting to note that the things have changed. While with the previous stations we had better results with the empty class (obvious, since they usually were in that state), with these stations the classifier perform better with the not empty class, reaching a recall of 83% and a f1 score of 90%.
However, the empty has good results too, with a good f1-score of 60%, mainly due to the perfect recall score. Each time a station is empty, the classifier manages to recognize that situation, and correctly assign the empty label.

3.4.5.3 Threshold tuning

Like we did in the previous experiments, I wanted to check what happens at the variation of the threshold in this case too. The results in the various scenarios are obtained with the same parameters described in the previous chapter.

Threshold	P(e)	R(e)	F1(e)	P(ne)	R(ne)	F1(ne)
1	0,44	1	0,6	1	0,83	0,9
2	0,57	1	0,72	1	0,73	0,84
3	0,66	1	0,79	1	0,66	0,77

Table 3.18 Statistics of less frequent stations with threshold tuning

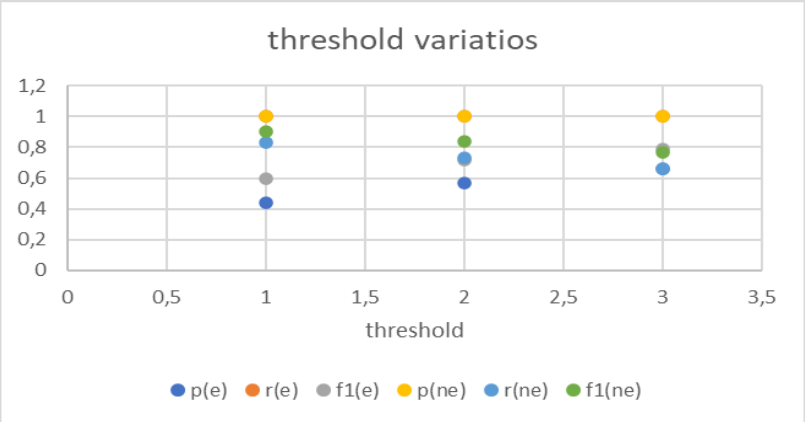

Figure 3.10 Quality of predictions at variation of threshold

It is noteworthy that even for these stations, increasing the threshold makes the classifier perform better for the empty class, and worse for the not empty class. Once more, we can desume that allowing more events to be classified as critical, helps the predictor in telling that a station will be in that situation. When the threshold is set to 3, actually, the classifier performs even slightly better with the empty class, in an unexpected trend inversion.

About those perfect scores in the recall of empty class and precision of non empty class, they are most likely due to how the prediction works: as we saw in Chapter 2, when testing the classifier, for each testing window I scan through all the rules I obtained, and only if I do not find a rule that matches with that window, I predict a not empty state. Since the amount of extracted rules is very large, it is very unlikely that I will not find a rule about a window, and when it happens, the prediction of normal state is most likely correct.

By the way, from these results, it is clear that, for more "normal" stations, it is acceptable to discard and not to consider those stations that are often empty.
Merging these results, with the ones obtained about the more frequent stations, we can say that the classifier is behaving even better than expected.

3.4.6 Comparisons with the baseline

At this point we are nearly at the end of the analysis. For this reason, we decided to try and understand if our classifier was performing better than a given baseline. It would be useful to understand if the complexity of our process is actually worth it, or if it does not produce a significant improvement.

3.4.6.1 The baseline

The first thing we need is to define our baseline. The most immediate way it comes in mind is to write a simple classifier that, given a set of records, predicts that at the next timestamp, the station of that record will be in the same state indicated by the record itself.
An example should clarify this. If we have the following record:

```
16    15-05-2008 12:00:00 15   1
```

that tells us that station with id = 16, at twelve o'clock of 15th May 2008 had 15 free slots, and 1 used slot (meaning that it is in an empty critical state), then our simple classifier will predict that at 1 PM o'clock that station will still be empty.
In the testing phase, the code will just check if this statement is true, by checking the record relative to that time.

I wrote a very simple Java application that performs this process, called DumbExtractor, and that I already described previously in this chapter.

3.4.6.2 Results of comparison with the baseline

The good thing about this baseline is that it is easily executable on each station, even on those which are almost impossible to take into account with my classifier.
Moreover, it performs very fast, since there is little to none data processing, as we have already seen.
I will report here 3 significative examples of what this baseline is able to do.

Station frequently empty
A station like 226, which is one of those unpopular station, is usually in the critical empty state. I report the results obtained by the baseline for this station.

P(E)	R(E)	F1(E)	P(NE)	R(NE)	F1(NE)
0,81	0,87	0,84	0,38	0,29	0,33

Table 3.19 Baseline with "*often empty"* stations

As expected, in cases like this, were the station is often empty, the baseline performs excellently with empty class, but poorly with the not empty class.

Stations frequently not empty
In this case, instead, we take into account a station like the one with id 154. This station is practically always in a not empty state. These are the results obtained with the baseline:

P(E)	R(E)	F1(E)	P(NE)	R(NE)	F1(NE)
0,19	0,22	0,21	0,95	0,94	0,94

Table 3.20 Baseline with *"often not empty"* stations

We have the opposite situation, where the baseline miserably fails to assign correctly the empty labels (due to the fact that in those rare cases in which the station is empty, the following timestamp it is again not empty), while it is nearly perfect in predicting the not empty states.

Balanced stations
Eventually, stations like the 126th are more or less 50% of the time empty, 50% of the time not empty. The results obtained for such a station by the baseline are:

P(E)	R(E)	F1(E)	P(NE)	R(NE)	F1(NE)
0,73	0,76	0,75	0,69	0,65	0,67

Table 3.21 Baseline with *"balanced"* stations

In this case, the baseline is performing very well in both cases. It manages to predict correctly the empty state 3 times out of 4, and with the not empty state it performs similarly good.

The average performance of this baseline, computed based on **all** the 284 stations of the file is shown in the table 3.20, while the difference of performance of my proposed solution and the baseline is indicated by table 3.21.

P(E)	R(E)	F1(E)	P(NE)	R(NE)	F1(NE)
0,42	0,47	0,45	0,82	0,79	0,82

Table 3.22 Average performance of the baseline

ΔP(E)	ΔR(E)	ΔF1(E)	ΔP(NE)	ΔR(NE)	ΔF1(NE)
+0,11	+0,53	+0,24	+0,18	-0,08	-0,02

Table 3.23 Differences between proposed solution and baseline

As we can see from table 3.20, the results of the baseline about the not empty class are very good. This is because in the data set we have a lot of stations (about a half) that are almost always in a normal state. The baseline manages to handle them correctly and to give us good results.
Instead, the empty class predictions are actually lacklustre. Those particular stations always in the empty state contribute to increase the average values, but actually, if we removed them from the data set, we would obtain much worse results.

What it is clear by looking at table 3.21 is that using the classifier we are discussing in these experiments improves a lot the results. Taking into account frequent patterns in the behaviour of the stations, and use these patterns to try and predict the future gives better result than just assuming that the station will be in the same state for two consecutive time instants. The small flaws in the results about “not empty” class are due to those stations that are almost always in a “not empty” state, that I could not take into account while using my classifier, since I could not extract any rules about them. By the way, such a classifier would always predict “not empty” state for these stations, obtaining results similar to the baseline.

Lastly, the fact the baseline works better when there are 50:50 cases, is probably due to the lack of correlation among states of such stations. In fact, using rules for such stations becomes, from a probability point of view, irrelevant.

3.4.7 Introduction of the distance constraint

As my last experiment with the data set, I decided to introduce a new constraint, that could make the computation less heavy without impacting too much on the results. It is something similar to the introduction of the sampling time interval, but while that was absolutely necessary to make the classifier work, some other constraints may just be accessories to improve the process.

After thinking about the data I was given for my experiments, I took the decision to use the information stored in the already described *stations.csv* file to perform some heuristics.

In particular, thanks to the information about the position of the various stations, it is possible to compute the distance among them. This is done by means of the Pythagorean Theorem. In fact, the stations we consider are all in Barcelona (in other words, very near between each other), and for this reason we can neglect the curvature of the Earth.
So, to know the relative distance of two stations, we extract the square root of the sum of the squares of the differences of latitude and longitude.

Thanks to the information about relative distance, we can, indeed, perform some filtering. If we suppose that we are performing predictions for intervals of 30 minutes, in fact, we can assume that if a station is in a place that is so far not to be reachable in that interval of time, we can discard their relative relationships.
It is similar to what we have done with “Banned stations”, but with a different principle to apply the filter.

3.4.7.1 Procedure

As we already said, to perform this experiment I used a modified version of TestRules, called TestRulesWithDistance. This code is not very different, since it only takes one more parameter (the path of stations.csv), and use the information stored there to perform an additional filter.

In particular, it extract from the file the latitude and the longitude of the station we are testing, and then it creates a list of “accepted” stations by computing the distances with each one of them.

```
Read lat and long of Station;
Foreach station:
      Read lat and long of station;
      Compute distance with Station;
```

```
        If(distance <= maxDistance)
             Store station in a list;
        Endif
End foreach
```

Pseudo-code to select the near stations

At this point we have a collection that is a white-list of stations. If a record refers to a station not present in this white list, that record is discarded, because we consider it not influent for the chosen station.

```
 Foreach record
        Read station
        If (whitelist does not contain station)
             Filter record;
        Endif
End foreach
```

Pseudo-code to filter the distant stations

After this we can proceed normally, like we do in TestRules. In fact, if we do these two passage, we will only take into account those records that refer a near station.

3.4.7.2 Results of the classifier with the distance constraint

At this point, we can see and discuss the results achieved by introducing this constraint.
When I worked with this version of the classifier, I tried tuning not only the distance, but also the interval (that can have a impact on the distance!) and on the threshold.

Sampling interval and maximum distance actually have a strong relationship between them because increasing the interval makes less acceptable using short distance: in two hours I can easily ride for 1 km, and using this constraint would be not very correct.

The maximum distance values I used are 1km, 2km and 5km (very similar to not introducing any constraint).
The interval values I used are 30 minutes, 1 hour and 2 hours.
The threshold values are once again 1, 2 and 3.
Also in this case I am using the binary version of the classifier, and a window of size = 2, so to keep the experiment as simple as possible.

Distance variation

This results were obtained by keeping a fixed value for the interval (1 hour) and for the threshold (2).

Distance	P(e)	R(e)	F1(e)	P(ne)	R(ne)	F1(ne)
1	0,64	0,62	0,6	0,73	0,62	0,62
2	0,65	0,74	0,67	0,59	0,46	0,48
5	0,67	1	0,72	1	0,73	0,84

Table 3.24 Classifier performance at the variation of the distance constraint

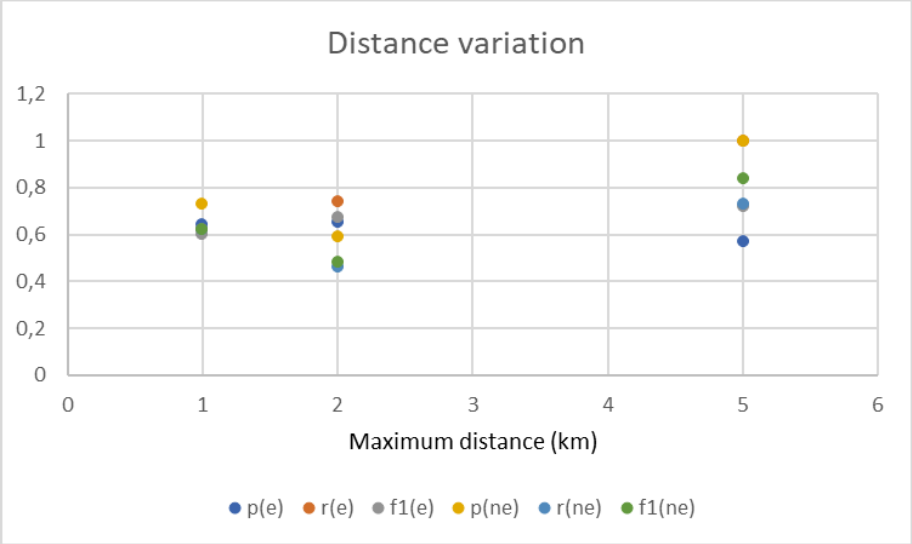

Figure 3.11 Quality of prediction at the variation of the distance

These are the results obtained when tuning the distance parameter. We can see how the classification about the empty labels becomes better and better at the increasing of the distance value. This means that with a value of 1 km we are probably filtering out important stations that allow us to predict correctly future empty states.
It is interesting to notice the weird thing that happens with the Not Empty class, on the other hand. For some reason, increasing the maximum allowed distance worsen the results. This may be due to the fact that introducing more stations lead to have more rules, and to have more classifications to the empty class. By the way, it is hard to explain why with the maximum distance, the results are way better.

Threshold variation

These results are obtained by keeping a fixed value for interval (1 hour) and a fixed value for distance (1 km).

Threshold	P(e)	R(e)	F1(e)	P(ne)	R(ne)	F1(ne)
1	0,5	0,27	0,31	0,75	0,87	0,8
2	0,64	0,62	0,6	0,73	0,62	0,62
3	0,68	0,69	0,66	0,59	0,53	0,53

Table 3.25 Classifier performance at the variation of the threshold

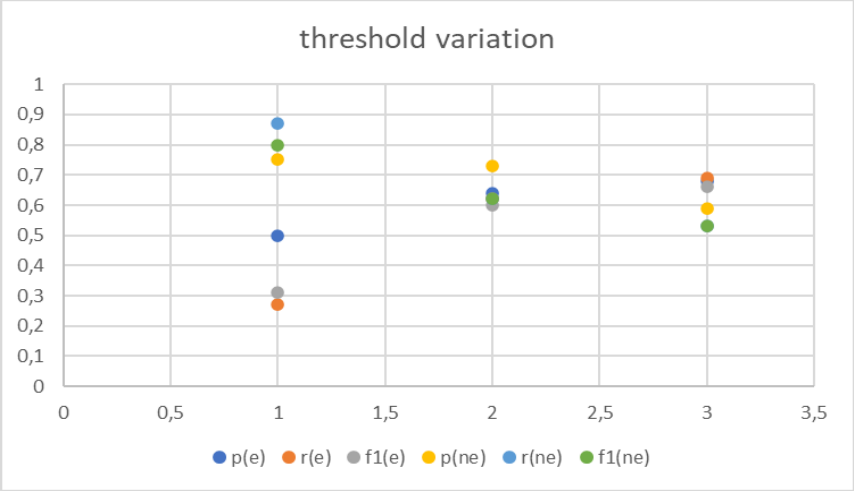

Figure 3.12 Quality of prediction at the variation of the threshold

The behaviour of this classifier at the variation of the threshold is exactly the same of the one without distance constraints. If we decrease the threshold, it becomes more difficult to predict correctly events in the empty class, while the ones in the not empty class becomes more easily detectable.

Interval variation

These results are obtained by keeping a fixed value for threshold (2) and a fixed value for distance (1 km).

Interval	P(e)	R(e)	F1(e)	P(ne)	R(ne)	F1(ne)
1	0,64	0,62	0,6	0,73	0,62	0,62
1,5	0,67	0,58	0,6	0,64	0,61	0,56
2	0,59	0,4	0,46	0,58	0,6	0,58

Table 3.26 Classifier performance at the variation of the sampling interval

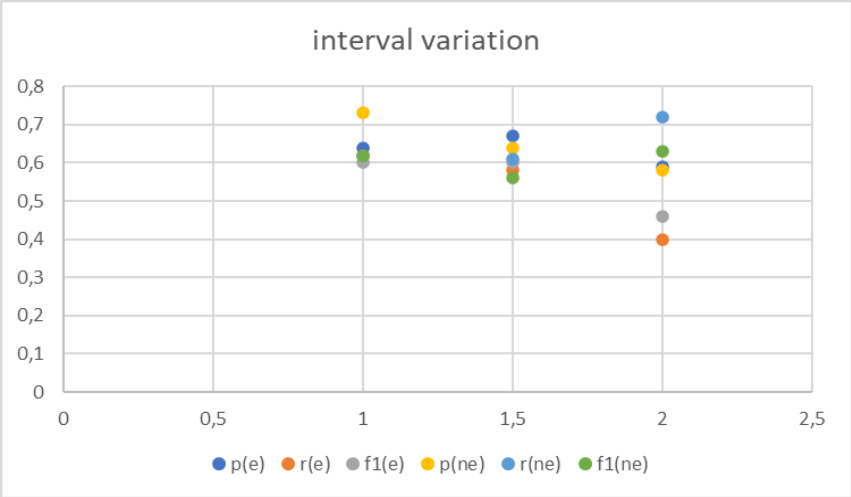

Figure 3.13 Quality of prediction at the variation of the interval

Increasing the time interval has two effects: it makes relationships between evens looser (since the events are more distant in time, they probably affect each other less), and makes the distance constraint less acceptable.
This is why we see a general decrease of performances in these results.

Chapter 4
Conclusions

In the light of the results of this work, we can conclude that exploring new ways to solve a problem that has been faced already by many studies is still promising. The problem is complex especially due to the amount of data that has to be analysed, and improving efficiency and quality is always an open challenge.
In the particular case of this thesis, it is clear that taking into account the information about time and space relationships among events is crucial to obtain better results. Furthermore, it is possible to introduce constraints that allow for a faster execution of such a heavy process. It is, in fact, very important to transform a complex and big data set into a simpler and more compact piece of information, without losing expressive power.

The results obtained during the testing phase met the expectations. In some case they were even better than what you could think, in particular when using very small window size to predict events near in time. This showed and told that a classifier built in this way gives better results with nearer events, respecting the logic for which the farther two happenings, the less related they are.

It was, then, confirmed the importance of properly calibrating the support parameter, which was one of the biggest challenges in this work. It was very hard to find ways to be able to lower that threshold, in order not to filter out too much, but it is definitely possible by means of interval sampling and adding distance constraints.

Plus, it was clear that using a binary predictor for very heterogeneous data sets can lead to much better results than using a single ternary one.

Furthermore, the definition of the classes themselves proved to be extremely important for such a classifier. Making a class more or less including has a huge impact on the quality of the predictions, and a trade-off between meaningfulness and performances must be found.

It was, instead, delusional how weird were the results with the limitations on the space. It is true that the particular dataset on which experiments were conducted was probably not the

most suitable for this purpose (since bike stations in Barcelona are not enough faraway), but in some cases it looked like there was a lack of correlation that was very difficult to explain.

However, from the comparison with the results obtained with the baseline, in all the different cases that were analysed, it is clear that this new approach brought to an important improvement in the field of association rule-based classification, and that continuing working on this may lead to important results.

Something that can and should be explored in detail is the possibility to extract space patterns independent from absolute position of the events. It would be, indeed, interesting to know if events that happens with a given relative space disposition, are likely to happen (with the same disposition) even elsewhere. Starting from the work presented in this thesis, this should not be too difficult to test.

To sum up, exploring new possibilities in the field of classification, and in particular in association rule-based prediction, is a rewarding work, that still presents ahead lots of challenging issues. This project, in fact, highlighted how a simple heuristic consideration made on the data to analyse or the process to use for analysis can open to big chances of improving what already exists and seems immutable.