Master's Degree in
Computer Engineering

Master's Degree Thesis

Associative classification on spatio-temporal sequences

Supervisors
prof. Paolo Garza, Politecnico di Torino
prof. Abolfazl Asudeh, University of Illinois at Chicago

Candidate
Niccolò Spagnuolo

Academic Year 2020-2021

Abstract

The main purpose of the study is to build a system to perform associative classification on spatio-temporal sequences. The proposed methodology is composed of four ordered phases: preprocessing, frequent itemsets mining, association rules generation and prediction model training. The model presented is eventually compared to other state-of-the-art classification algorithms such as Decision Trees, Random Forests and Support Vector Machines. On balance, the prediction model achieves a higher precision for the critical and most rare class with respect to its competitors.

Acknowledgements

To begin with, I would like to thank professor Paolo Garza and professor Abolfazl Asudeh for their support and availability throughout the thesis work.

Moreover, I have to mention Jenna Stephens and thank her for being supportive and prompt to resolve any issues that came up related to the academic program.

Then, I would love to thank my closest group of friends with whom I shared the last six years and I sincerely hope more to come: Luca, my coffee-mate, Angelo, the one-liner man, Fabrizio, the best tour operator in the field and last but not least Michele, the most stylish cook I have ever known.

Next, I mention Alessandro and Giuseppe for being my brothers, not by blood, since always.

Finally, I thank from the bottom of my heart my family for being always there when I needed and cheerful every day of my university career. In particular, I mention my sister Annachiara, my cousins Ciu and Francesco for their unconditional support during difficult times.

Contents

List of Tables **5**

List of Figures **6**

1 Introduction **9**

2 Related Work **11**
2.1 Association rules mining . . . 11
2.1.1 Frequent itemsets mining . . . 11
2.1.2 The Apriori algorithm . . . 12
2.1.3 The FP-Growth algorithm . . . 13
2.1.4 The PFP algorithm . . . 14
2.1.5 Association rules generation . . . 15
2.2 Sequential pattern mining . . . 15
2.3 Classification . . . 16
2.3.1 Associative classification . . . 16
2.3.2 Decision Trees . . . 17
2.3.3 Random Forest . . . 18
2.3.4 SVM . . . 19
2.3.5 Model evaluation . . . 20
2.4 Apache Spark . . . 22

3 The proposed methodology **27**
3.1 Problem statement . . . 27
3.2 Preprocessing . . . 27
3.2.1 Filtering . . . 28
3.2.2 Alignment . . . 28
3.2.3 Sliding-window transformation . . . 29
3.3 Frequent itemsets mining . . . 30
3.4 Association rules generation . . . 32
3.4.1 Post processing filter . . . 32
3.5 Prediction model training . . . 33
3.5.1 Example . . . 34

4 Experimental results **35**
4.1 Dataset analysis . . . 35
4.2 Parameters effect on association rule generation . . . 37
4.2.1 Testing environment . . . 38
4.2.2 Radius . . . 38
4.2.3 MinSup . . . 39
4.2.4 Window size . . . 40
4.2.5 Granularity . . . 41
4.3 Prediction model training . . . 42
4.3.1 Testing environment . . . 42
4.3.2 Categories of event . . . 42
4.3.3 Timeslots . . . 43
4.3.4 Tested classifiers . . . 43

5 Conclusions **55**

List of Tables

2.1 MAIN RDD TRANSFORMATIONS IN SPARK 23
2.2 MAIN RDD ACTIONS IN SPARK 24
2.3 MAIN PAIRRDD TRANSFORMATIONS IN SPARK 25
2.4 MAIN PAIRRDD ACTIONS IN SPARK 25
3.1 INPUT DATASET EXAMPLE 28
3.2 INPUT DATASET EXAMPLE AFTER ALIGNMENT 29
3.3 INPUT DATASET EXAMPLE AFTER SLIDING-WINDOW TRANSFORMATION 29
3.4 TEST SET TRANSACTIONS FOR CLASSIFICATION EXAMPLE . . . 34
3.5 ORDERED ASSOCIATION RULES FOR CLASSIFICATION EXAMPLE 34
4.1 SAMPLE OF RECORDS AND EXTRACTION OF EVENTS 43

List of Figures

2.1 SVM example . . . 20
2.2 Confusion matrix for a binary classifier . . . 21
3.1 Phases of the proposed methodology . . . 27
4.1 Locations of bike sharing stations . . . 35
4.2 Events per station ECDF thr = 0 . . . 36
4.3 Events per station ECDF thr = 0 . . . 36
4.4 Events per station ECDF thr = 1 . . . 36
4.5 Events per station ECDF thr = 1 . . . 36
4.6 Events per station ECDF thr = 2 . . . 37
4.7 Events per station ECDF thr = 2 . . . 37
4.8 Events per station ECDF thr = 3 . . . 37
4.9 Events per station ECDF thr = 3 . . . 37
4.10 Execution time by *radius* . . . 39
4.11 Number of itemsets by *radius* . . . 39
4.12 Number of rules by *radius* . . . 39
4.13 Distribution of rules confidence . . . 39
4.14 Execution time by *minSup* . . . 40
4.15 Number of itemsets by *minSup* . . . 40
4.16 Number of rules by *minSup* . . . 40
4.17 Distribution of rules confidence . . . 40
4.18 Execution time by *window* . . . 41
4.19 Number of itemsets by *window* . . . 41
4.20 Number of rules by *window* . . . 41
4.21 Distribution of rules confidence . . . 41
4.22 Execution time by *gran* . . . 42
4.23 Number of itemsets by *gran* . . . 42
4.24 Number of rules by *gran* . . . 42
4.25 Distribution of rules confidence . . . 42
4.26 AEAFT classifier metrics by AEAFT_minSup . . . 44
4.27 AFT, AF classifiers metrics by AFX_minSup_ruleLen_minConf . . . 45
4.28 AFT classifier metrics by AFT_minSup_ruleLen . . . 46
4.29 AET, AE classifiers metrics by AEX_minSup_ruleLen . . . 47
4.30 BFT, BF classifiers metrics by BFX_minSup_ruleLen . . . 48
4.31 BET, BE classifiers metrics by BEX_minSup_ruleLen . . . 49
4.32 Double binary classifiers metrics by XXX_minSup+XXX_minSup . . . 50

4.33 Competitor classifiers metrics 51
4.34 Comparison with competitor classifiers metrics 51
4.35 Mixed approach DT+AFT classifiers metrics by DT+AFT_minSup 52
4.36 Mixed approach AFT+DT classifiers metrics by AFT_minSup+DT 53
4.37 Associative classifiers comparison 54

Chapter 1

Introduction

Associative classification is a data mining approach that integrates association rule mining and classification to build classifiers. The structure of an associative classifier consists of two steps. First, association rules are extracted from the training data. Then, pruning techniques are applied to select a subset of high-quality rules, by applying support and confidence based pruning.

The target of this work is to define a methodology to perform associative classification on spatio-temporal sequences. The proposed methodology is composed of four ordered phases: preprocessing, frequent itemsets mining, association rules generation and prediction model training. The preprocessing stage is entitled to convert input time-series data to a set of transactions. Such transactions are the input of the frequent itemsets mining stage, accomplished by a parallel implementation of FP-Growth for Apache Spark, in which a spatial constraint is added to prune the search space. Afterwards, given the frequent itemsets, association rules are generated and, finally, the prediction model is trained to conduct associative classification.

This thesis is organized as follows. Chapter 2 (*Related work*) provides an overview of the main background concept the thesis takes advantage from. Chapter 3 (*The proposed methodology*) describes in detail the system steps aforementioned. Chapter 4 (*Experimental results*) presents the results of the proposed methodology on an example dataset. Moreover, such results are compared to the ones obtained by competitor classification algorithms such as Decision Trees, Random Forests and Support Vector Machines. Finally, Chapter 5 (*Conclusions*) draws some general conclusions.

Chapter 2

Related Work

This chapter is going to highlight the main background concepts that this Thesis takes advantage from. To begin with, association rule mining is presented, then the second section briefly describes sequential pattern mining. Furthermore, the third section defines the revision of classification and, finally, the Apache Spark data processing library is outlined since it has been used in the development of the proposed model.

2.1 Association rules mining

Mining of association rules is one of the most important data mining tasks. Its main objective is to extract frequent correlations or patterns from a transactional database. It was firstly introduced by Agrawal et al. [1] in 1993 and since then, several algorithms and extensions have been released. Its classic application is in the market-basket data analysis, in which it is used to find co-occurrence relationships among the items purchased by customers.

An example of association rule is the following:

$$Diapers \Rightarrow Beer\ [Support = 2\%, Confidence = 30\%] \tag{2.1}$$

The rule says that 2% of customers buy diapers and beer together and, those who buy diapers also buy beer the 30% of times. These two metrics are *Support* and *Confidence* respectively, which are going to be defined in Section 2.1.5. The problem of rule mining can be decomposed into two subproblems:

- Extraction of frequent itemsets.
- Generation of association rules from frequent itemsets.

2.1.1 Frequent itemsets mining

Let $T = \{T_1, ..., T_n\}$ be a transactional database where the database schema $S = \{A_1, ..., A_m\}$ consists of a large number of attributes and the attribute domains are binary, that is, $dom(A_i) = \{0, 1\}$. The attributes can be seen as properties that an instance does have or does not have. Let $U = \{i_1, i_2, ..., i_m\}$ be a set of properties or *items*. A **transaction**

$(t_1, ..., t_m) \in dom(A_1) \times ... \times dom(A_n)$ from our transactional database T with schema S, can be reformulated as an **itemset** I by $i_i \in I \iff t_i = 1$. We call a set $Z \subseteq I$ an **association**. The **support** of an arbitrary itemset S is the probability $P(S)$ of observing S in a randomly selected record of T.

Definition 1 (Support) *The support of an itemset S is the fraction of transactions that contain S as a subset.*

$$sup(S) = \frac{|\{\forall T_i \in T | S \subseteq T_i\}|}{|T|} \tag{2.2}$$

An itemset is *frequent* when its support is greater than or equal to a *minSup* threshold. Extracting all possible combinations of items could be very computationally expensive, given that in a universe U there are $2^m - 1$ possible itemsets. Frequent itemsets mining deals with the extraction of frequent itemsets by exploiting different techniques: level-wise approach such as *Apriori* [2] or approaches without candidate generation such as *FP-growth* [3].

2.1.2 The Apriori algorithm

The Apriori algorithm [2] is based on two properties: the first one is the *support antimonotone property* which states that "if an itemset S is contained in a transaction T_i, also any subset J of S must be contained in the same transaction T_i. Thus, the support of J will be always greater than or equal to the support of S". The second one is the *downward closure property*, which is a consequence of the first one, stating that: "if an itemset is frequent, then all its subsets must also be frequent". The Apriori algorithm is a level-based approach, which means that at each iteration it extracts itemsets of a given length k. Two main steps for each level are executed:

- **Candidate generation**
  1. Join step: generate candidates of length $k + 1$ by joining frequent itemsets of length k;
  2. Prune step: apply the *downward closure property* by pruning length $k + 1$ candidate itemsets that contain at least one k-itemset that is not frequent.
- **Frequent itemset generation**
  1. Scan database to count support for $k + 1$ candidates;
  2. Prune candidates below *minSup*.

Given that L_k is a frequent itemset of length k, the candidate generation is performed as follows:

1. Sort l_k candidates in lexicographical order;
2. for each candidate of length k:
   (a) Self-join with each candidate sharing same L_{k-1} prefix;
   (b) Prune candidates by applying the *downward closure property*.

This algorithm has several bottlenecks: first of all, during the count of support for candidates, it scans the transaction database to count support of each itemset. This can be computationally expensive when the total number of candidates can be large or if one transaction may contain many candidates. Another pain-point is the candidate generation: candidate sets might be huge, in particular, 2-itemset candidate generation is the most critical step and, additionally $n + 1$ scans when longest frequent pattern length is n. The major factors affecting performance of Apriori are:

- Minimum support threshold: lower support threshold increases number of frequent itemsets, leading to a larger number of candidates and larger (max) length of frequent itemsets;
- Dimensionality (number of items) of the data set: more space is needed to store support count of each item and, if number the of frequent items also increases, both computation and I/O costs may also increase;
- Size of the database: since Apriori makes multiple passes, run time of the algorithm may increase with the number of transactions;
- Average transaction width: transaction width increases in dense data sets, as well as increasing max length of frequent itemsets, and the number of subsets in a transaction increases with its width.

2.1.3 The FP-Growth algorithm

The *Frequent-Pattern Growth* [3] algorithm is another method used to extract frequent itemsets, and it is based on a main memory compressed representation of the database (the *FP-tree*). This compression is suitable for dense data distributions, less so for sparse ones. Frequent pattern mining is made by means of recursive visits of the *FP-tree* and by applying a *divide-et-conquer* approach. Only two database scans are needed: the first one to compute items support and the second one to build the *FP-tree*. The algorithm structure is as follows:

1. *FP-tree* construction:
   (a) count items support and prune items below the *minSup* threshold;
   (b) build the *Header Table* by sorting items by a decreasing *support* order;
   (c) create the *FP-tree*, for each transaction t:
      i. order transaction t items in decreasing support order;
      ii. insert transaction t in *FP-tree* by using existing path for common prefix and creating a new branch when path becomes different.
2. Scan the *Header Table* from lowest *support* item up:
   (a) for each item i in the *Header Table* extract frequent itemsets including item i and items preceding it in the *Header Table*:
      i. Build *Conditional Pattern Base* for item i (i-CPB) by selecting prefix-paths of item i from *FP-tree*;

ii. Recursive invocation of *FP-Growth* on i-CPB.

3. *FP-Growth* returns frequent itemsets and their support.

A parallel implementation of *FP-Growth* has been defined by Li et al [4] and it is a solution based on the *MapReduce* paradigm. A version built for *Apache Spark* would be exploited and modified throughout the Thesis to work for sequences and adding to it an online filter such as a spatial constraint to prune the algorithm search.

2.1.4 The PFP algorithm

The *Parallel FP-Growth* algorithm [4] takes advantage from a *MapReduce* approach to parallelize *FP-Growth* which intelligently *shards* a large-scale mining task into independent computational tasks and maps them onto *MapReduce* jobs. PFP can achieve near-linear speedup with capability of restarting from computer failures.

Given a transaction database *DB*, PFP acts in five steps, including three *MapReduce* phases:

1. **Sharding**: dividing the *DB* into successive parts and store the parts in P computers. Each part is called a *shard*.

2. **Parallel Counting**: it consists of a *MapReduce* job to count the *support* of all items that are in the *DB*. Each mapper inputs one shard of the *DB*. This step implicitly discovers the items' vocabulary I. The result is stored in F-list.

3. **Grouping items**: dividing all the $|I|$ items on F-list into Q groups. The list of groups is called *group list* (G-list) and each group is given a unique *group id* (gid).

4. **Parallel FP-Growth**: This step takes one *MapReduce* job where the map and reduce perform two different functions:

   (a) *Mapper - generating group-dependent transactions*: the mapper reads the G-list as a hash map which maps each item into the corresponding gid. The mapper instance is fed with a shard of the *DB*, for each T_i two steps are performed:

      i. For each item $a_j \in T_i$, a_j is replaced by the corresponding gid;

      ii. For each gid, if it appears in T_i, locate its rightmost appearance L, and output a key-value pair $\langle key = gid, value = \{T_i[1], ..., T_i[L]\}\rangle$, where the value is a generated group-dependent transaction.

   (b) *Reducer - FP-Growth on group-dependent shards*: when all mappers are finished, for each gid all corresponding group-dependent transactions are grouped into a shard of group-dependent transactions. Each reducer is assigned to process one or more group-dependent shard. For each shard a local FP-tree is built and its conditional FP-trees are recursively grown. During the recursion, it may output patterns.

5. **Aggregating**: the results in step 4 are aggregated with a *MapReduce* job to produce the final result.

2.1.5 Association rules generation

An association rule is an implication of the form: $X \Rightarrow Y$ where $Z = X \cup Y \subseteq U$ and $X \cap Y = \emptyset$. X and Y are a set of items, called an **itemset**, they are respectively the *antecedent* and the *consequent* of the association rule. The rule quality metrics are *support* and *confidence*.

Definition 2 (Rule support) *The support of a rule is the fraction of transactions in T that satisfy the union of items in the consequent and the antecedent of the rule.*

$$sup(X \Rightarrow Y) \coloneqq sup(X \cup Y) \tag{2.3}$$

On the other hand *confidence* can be seen as the conditional probability that a transaction contains Y given that contains also X, providing us the strength of the correlation.

Definition 3 (Confidence) *The confidence of a rule is defined as the fraction of itemsets that support the rule among those that support the antecedent. It can be seen as the frequency of Y in transactions containing X.*

$$conf(X \Rightarrow Y) \coloneqq P(Y|X) = \frac{sup(X \cup Y)}{sup(X)} \tag{2.4}$$

The target of generating association rules is to extract rules that have *support* and *confidence* greater than or equal to user specified minimum *support* (*minSup*) and minimum *confidence* (*minConf*). After having extracted frequent itemsets, the extraction of association rules is more straightforward. To generate rules for the frequent itemset f we use all its non-empty subsets. For each subset α, we output a rule $f - \alpha \Rightarrow \alpha$ if its *confidence* is greater than or equal to *minConf*.

2.2 Sequential pattern mining

Sequential pattern mining consists in discovering interesting subsequences in a set of sequences, where its interestingness can be measured such as its support. There are many real-life applications, whenever data can be encoded as sequences, such as market-basket analysis [5], text analysis [6], webpage click-stream analysis [7], e-learning [8] or bioinformatics [9]. Sequential pattern mining is designed to be applied to sequences, but it can be also applied to time-series after a preprocessing phase that uses discretization techniques. Sequential pattern mining algorithms can be divided into two categories: *breadth-first search* based or *depth-first search* based.

The *breadth-first search* algorithms such as GSP [5] proceed as follows: first the database is scanned to find frequent 1-sequences then, 2-sequences are generated by performing extensions on 1-sequences and so on. This approach is called *level-wise* since patterns are generated in ascending length order. This methodology could result in a very large search space as there are several ways to combine items to generate a potential sequential pattern.

Depth-first search algorithms such as SPADE [10], PrefixSpan [11], Spam [12], Lapin [13], CM-Spam [14], and CM-Spade [14] on the other hand, explore the search space in a different order. They start from sequences containing one item and then recursively perform extensions to produce larger sequences. Then, when a pattern can no longer be

extended, the algorithm backtracks to generate other patterns using other sequences. For these algorithms the application of the *downward-closure property* is crucial. This can greatly reduce the search space of sequential patterns.

2.3 Classification

Given a collection of class labels, and a collection of data objects labeled with a class label (**training set**) the definition of classification is to find a descriptive profile of each class, which will allow the assignment of unlabeled objects to the appropriate class. The classification model is validated on the **test set**, which is a collection of labeled data objects different from the **training set**.

In this section we are going to describe some classification algorithms, starting from the Associative classification on which our proposed methodology is based, and then providing an overview of competitor algorithms, such as Decision Trees, Random Forest and Support Vector Machines (SVM).

2.3.1 Associative classification

Associative classification was firstly introduced by Liu et al. in 1998 [15] and aims to integrate association rule mining and classification rule mining.

The model used to classify consists of association rules, they can be seen as a set of "if-then" clauses: if the current record matches the *antecedent* of the rule, it is then labeled according to the class of the *consequent* of the rule.

The model generation is composed of:

- Rule selection and sorting: based on confidence, support and correlation thresholds;
- Rule pruning: the training set is covered by selecting topmost rules according to previous sort.

The advantages of associative classification are:

- Interpretable model;
- Higher accuracy than decision trees: when correlation among attributes is considered;
- Efficient classification;
- Unaffected by missing data;
- Good scalability in the training set size.

The disadvantages of associative classification are:

- Rule generation may be slow: it depends on the support threshold;
- Reduced scalability in the number of attributes: rule generation may become unfeasible.

2.3.2 Decision Trees

A decision tree uses a tree-like structure where each node represents a feature (or attribute), the branch represents a decision rule and each leaf node represents the class label. It learns to partition on the basis of the attribute value and partitions the tree in a recursive way. There are several algorithms dedicated to the model learning, one of the earliest is the *Hunt's algorithm*. The general structure of such algorithm is the following: let D_t be the set of training records that reach a node t then:

- if D_t contains records that belong to the same class y_t, then t is a leaf node and is labeled as y_t;
- if D_t is an empty set then t is a leaf node labeled as the default (majority) class, y_d (the class with the highest probability);
- if D_t contains records that belong to more than one class, select the "best" attribute A on which to split D_t and label the node t as A, then split D_t into smaller subsets and recursively apply the procedure to each subset.

To determine which is the best attribute to split the records at each node, we prefer attributes with homogeneous class distribution. In order to evaluate the class distribution we need measures of node impurity such as:

- GINI index;
- Entropy;
- Misclassification error.

GINI index for a given node t is the following:

$$GINI\,(t) = 1 - \sum_j \left[p\,(j|t)\right]^2 \tag{2.5}$$

where $p(j|t)$ is the relative frequency of class j at node t.

When a node p is split into k partitions (children), the quality of split is computed as the weighted sum of the indexes:

$$GINI_{split} = \sum_{i=1}^{k} \frac{n_i}{n} GINI\,(i) \tag{2.6}$$

where n_i is the number of records at child i and n is the number of records at node p.

Entropy at a given node t is:

$$Entropy\,(t) = -\sum_j p\,(j|t) \log_2 p\,(j|t) \tag{2.7}$$

where $p(j|t)$ is the relative frequency of class j at node t.

One entropy based computation is the *information gain* that follows this formula:

$$GAIN_{split} = Entropy\,(p) - \left(\sum_{i=1}^{k} \frac{n_i}{n} Entropy\,(i)\right) \tag{2.8}$$

where parent node p is split into k partitions and n_i is the number of records in partition i. It measures reduction in entropy achieved because of the split then, it is needed to choose the split that maximises the *information gain*.

Another entropy based computation is the *gain ratio* which adjusts *information gain* by the entropy of the partitioning. It is defined as follows:

$$GainRATIO_{split} = \frac{GAIN_{split}}{SplitINFO} \tag{2.9}$$

where

$$SplitINFO = -\sum_{i=1}^{k} k\frac{n_i}{n} \log_2 \left(\frac{n_i}{n}\right) \tag{2.10}$$

The *misclassification error* at node t is:

$$Error\,(t) = 1 - \max_i P\,(i|t) \tag{2.11}$$

The advantages of Decision Trees are:

- Inexpensive to build, low training time;
- Extremely fast at classifying unknown records;
- Easy to interpret for small trees;
- Accuracy is comparable to other classification methods for simple data sets

On the other hand, one disadvantage is that accuracy may be affected by missing data.

2.3.3 Random Forest

Random Forest is a supervised learning algorithm. It is an ensemble method of decision trees generated on a randomly split dataset. The collection of trees is the forest, in which the individual decision trees are generated using an attribute selection measure such as information gain, gain ratio or GINI index. Each tree is decorrelated, feature subsets are sampled randomly, therefore different features can be selected as the best attribute for the split. For the classification problem each tree votes and the majority class is chosen as the final result. Hence, the algorithm works in four steps:

- Select random samples from a given dataset;
- Build a decision tree for each sample and get a prediction from each one;
- Perform a vote for each predicted result;
- Select the one with most votes as the final prediction.

The advantages of Random Forest are:

- Higher accuracy than decision trees;
- Fast training phase;

- Robust to noise and outliers;
- Provides the relative feature importance, estimating which features are important for the classification.

One disadvantage of Random Forest is that result can be difficult to interpret compared to a decision tree, where it is possible to check a decision by following a path in the tree.

2.3.4 SVM

Support Vector Machines are a supervised learning technique which is used in several applications such as: Natural Language Processing (NLP), voice and image recognition and computer vision. It achieves the maximum effectiveness in binary classification problems. Its key idea is to find an hyperplane that better divides a data set in two classes. To begin with, we provide some definitions:

- **Hyperplane**: if we consider a classification task in two dimensions, the hyperplane is a line that classifies a data set having different class memberships. In three dimensions the hyperplane is a plane, with more than three dimensions it is generally called hyperplane;
- **Support Vectors**: they are the nearest points to the hyperplane, if removed or modified they alter the position of the hyperplane. Therefore, support vectors are considered the most relevant in the data set;
- **Margin**: it is defined as the distance between the nearest two support vectors (belonging to different classes) to the hyperplane.

The aim of SVM is to select a hyperplane with the maximum possible margin between support vectors in order to have a lower classification error. SVM follows these steps:

- Searches a linearly separable hyperplane. If it finds more than one, it selects the one with the highest margin in order to improve the accuracy of the model;
- If no hyperplane exists, SVM uses a non linear mapping to transform the training data in a higher dimension where it is possible to use linear separation, this technique is called kernel trick.

The kernel trick is based on transforming the input data into higher dimension, whenever it is not possible to determine a linearly separable hyperplane. The functions that perform this mapping are called **kernel functions**. The most popular ones are the following:

- **linear kernel**, defined as:
$$K(x_i, y_j) = x_i \cdot y_j \tag{2.12}$$
- **polynomial kernel**, defined as:
$$K(x_i, y_j) = (x_i \cdot y_j + c)^d \tag{2.13}$$
- **Radial Basis Function (RBF) kernel**, defined as:
$$K(x_i, y_j) = e^{(-\gamma ||x_i - y_j||^2)} \tag{2.14}$$

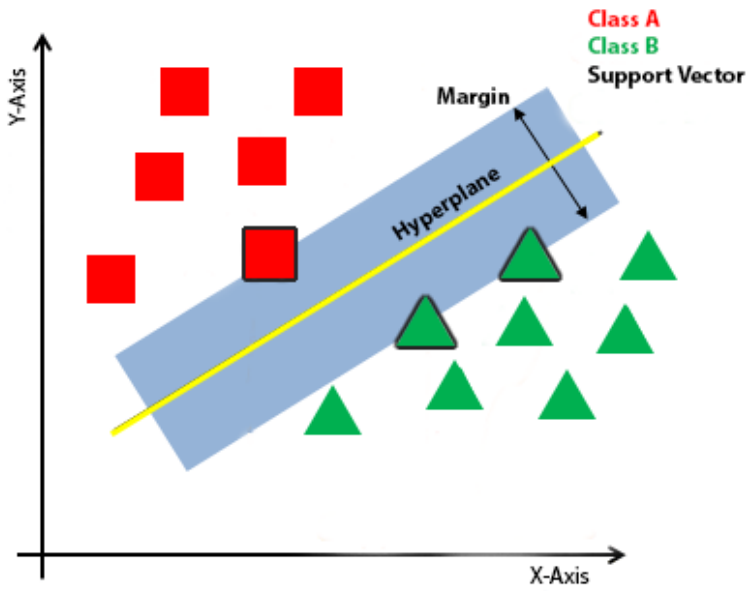


Figure 2.1: SVM example

The advantages of SVM are:

- Effective in high spatial dimensions;
- They treat non linear problems;
- More stable in the definition of parameters.

The disadvantages of SVM are:

- Non interpretable model;
- Non probabilistic method.

2.3.5 Model evaluation

The main metrics for performance evaluation of a model are:

- Accuracy;
- Precision;
- Recall.

In order to compute them it is necessary to build a *confusion matrix* which summarises the classification performance of a classifier with respect to the test set. For a binary classifier it is a table with two rows and two columns which reports the number of false

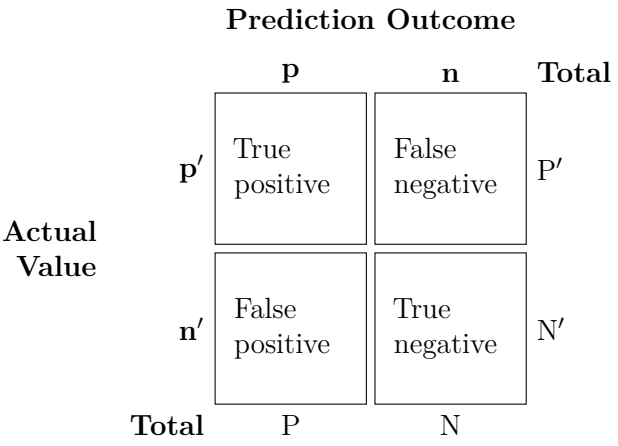


Figure 2.2: Confusion matrix for a binary classifier

positives, false negatives, true positives, and true negatives. Figure 2.2 shows an example of confusion matrix for a binary classifier.

Accuracy is computed as follows:

$$Accuracy = \frac{Number\,of\,correctly\,classified\,objects}{Number\,of\,classified\,objects} \tag{2.15}$$

for a binary classifier is:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.16}$$

However, accuracy is not appropriate for unbalanced class label distribution or for different class relevance.

Recall and *precision*, instead, are class specific measures and are defined as follows:

$$Recall(C) = \frac{Number\,of\,objects\,correctly\,assigned\,to\,C}{Number\,of\,objects\,belonging\,to\,C} \tag{2.17}$$

$$Precision(C) = \frac{Number\,of\,objects\,correctly\,assigned\,to\,C}{Number\,of\,objects\,assigned\,to\,C} \tag{2.18}$$

for a binary classifier, for the positive class are:

$$Recall(P) = \frac{TP}{TP + FN} \tag{2.19}$$

$$Precision(P) = \frac{TP}{TP + FP} \tag{2.20}$$

2.4 Apache Spark

Apache Spark is a unified analytics engine for large-scale data processing. In the context of Big Data it aims to achieve the following goals: *generality* in terms of diverse workloads, operators and job sizes, *low latency*, *fault tolerance* and *simplicity* by means of job scheduling and job synchronization.

Data in Spark is represented as *Resilient Distributed Datasets* (RDD) which contain an arbitrary collection of objects distributed across the worker nodes of clusters so that they can be processed in parallel. RDDs are stored in main memory whenever possible, otherwise on the local disk. In order to provide fault tolerance and to reduce synchronization costs among the nodes, RDDs are immutable: whenever there is the need to change content for a RDD, a new one is instantiated. Spark programs are written in terms of operations on RDDs.

The structure of Spark programs is based on a master/slave architecture. It has one coordinator, the *Driver*, which communicates with many distributed *worker* nodes. The *Driver* contains the main method, defines the workflow of the application, defines RDDs which are partitioned on the nodes of the cluster and invoke parallel operations on RDDs. It converts the user program into tasks which are the scheduled on the *executors*. *Executors* are the processes of *worker* nodes who are in charge of running different computations (tasks) in parallel. Each *executor* runs, on its own partition of the RDDs, the operations specified in the driver. Once they finish running, results are recombined after distributing them to the different nodes by means of the *shuffle* operation.

Two types of operations are available for RDDs: *transformations* and *actions*.

- *Transformations*, due to the immutability of RDDs turn a RDD into a new RDD, whose contents varies by applying the certain operation to the input RDD;
- *Actions* return a value to the *Driver* program or write the result in the storage, the latter could be large and in this case it is stored in the (distributed) file system.

The presence of an *action* throughout the *Driver* program is crucial because *transformations* in Spark are *lazily computed*, which means that the content of the new RDD are computed only in the moment an action is specified. When a *transformation* is invoked Spark keeps only track of the dependency between the input RDD and the new RDD. This ensures greater efficiency and reliability. Table 2.1 and Table 2.2 summarize the main transformations and actions.

A variant of the RDD is the *PairRDD*, which represents a collection of key-value pairs, they support specific operations related to data grouping. Table 2.3 and Table 2.4 summarize the main *PairRDD* transformations and actions.

Transformation	Description
Filter	It returns a new RDD containing only the elements of the input RDD that satisfy a user specified condition.
Distinct	It returns a new RDD containing the distinct elements of the input RDD.
Sample	It returns a new RDD containing a random sample of the input RDD with or without replacement.
Map	It creates a new RDD containing exactly one element y for each element x of the input RDD by applying a user defined function f on x.
FlatMap	It creates a new RDD by applying a function f on each element of the input RDD, the function f is applied on an element x of the input RDD and returns a list of values [y], where [y] can be the empty list.
MapToPair	It creates a new PairRDD by applying a function f on each element of the input RDD. The new RDD contains one tuple y for each element x of the input RDD.
FlatMapToPair	It creates a new PairRDD by applying a function f on each element of the regular input RDD. The new RDD contains a list of tuples [y] for each element x of the input RDD, where [y] can be the empty list.
ZipWithUniqueId	It creates a new PairRDD by coupling each element of the input RDD with a unique identifier.
Union	It returns a new RDD containing the union of the elements of the input RDD and the elements of the one passed as parameter to union(). Duplicates values are not removed. All the RDDs must have the same data type.
Intersection	It returns a new RDD containing the intersection of the elements of the input RDD and the elements of the one passed as parameter to intersection(). All the RDDs must have the same data type.
Subtract	It returns a new RDD containing the elements appearing only in the input RDD and not in the one passed as parameter to subtract(). All the RDDs must have the same data type.

Table 2.1: MAIN RDD TRANSFORMATIONS IN SPARK

Action	**Description**
COLLECT	It retrieves all the elements of the RDD within a local collection.
COUNT	It returns the number of elements of the RDD.
TAKE	It returns a collection of the first n elements of the RDD.
TAKESAMPLE	It returns a collection containing a random sample of size n of the RDD, with or without replacement.
TOP	It returns a collection containing the top n elements of the RDD based on the default sort order of the objects.
FIRST	It returns the first element of the RDD.
REDUCE	It returns a single object obtained by combining the values of the objects of the RDD by using a user provided function. The function must be associative and commutative. The object returned by the function and the object of the RDD belong to the same class.
AGGREGATE	It returns a single object obtained by combining the values of the objects of the RDD by means of a user provided function and an initial zero value. The function must be associative and commutative. The returned object can belong to a different class with respect to the object of the RDD.
SAVEASTEXTFILE	It stores the RDD as a text file, using string representations of elements, in a group of files in the given output path.

Table 2.2: MAIN RDD ACTIONS IN SPARK

Transformation	Description
ReduceByKey	It returns a new PairRDD containing one pair for each key of the input PairRDD. The value of each pair of the new PairRDD is obtained by combining the values of the input PairRDD with the same key. The input PairRDD and the new PairRDD have the same data type.
GroupByKey	It returns a new PairRDD containing one pair for each key of the input PairRDD. The value of each pair of the new PairRDD is a list containing the values of the input PairRDD with the same key.
MapValues	Apply a function over each pair of a PairRDD and return a new PairRDD. The applied function returns one pair for each pair of the input PairRDD. The function is applied only to the value without changing the key. The input PairRDD and the new PairRDD can have a different data type.
FlatMapValues	Apply a function over each pair of a PairRDD and return a new PairRDD. The applied function returns a set of pairs (from 0 to many) for each pair of the input PairRDD. The function is applied only to the value without changing the key. The input PairRDD and the new PairRDD can have a different data type.
SubtractByKey	It returns a new PairRDD where the pairs associated with a key appearing only in the input RDD and not in the one passed as parameter.
Join	It returns a new PairRDD corresponding to the join of the two PairRDDs. The join is based on the value of the key.

Table 2.3: MAIN PAIRRDD TRANSFORMATIONS IN SPARK

Action	Description
CountByKey	It returns a local map containing the number of elements in the input PairRDD for each key of the input PairRDD.
CollectAsMap	It returns a local map containing the pairs of the input PairRDD.

Table 2.4: MAIN PAIRRDD ACTIONS IN SPARK

Chapter 3

The proposed methodology

In this chapter, the system to perform associative classification on spatio-temporal sequences is presented.

The proposed methodology is composed of four phases as shown in Figure 3.1.

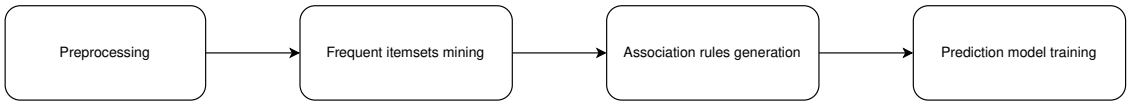


Figure 3.1: Phases of the proposed methodology

3.1 Problem statement

Given a dataset in which exists the concept of events associated with space and time features, in particular, the features of the elements of such dataset should be:

- A timestamp;
- The coordinates of a location;
- An event or a state.

The target is to train an associative classification model to predict future events/states.

3.2 Preprocessing

The preprocessing phase is entitled to convert input time-series data to a set of transactions, where the concept of sequentiality is included beneath the items of a transaction given a particular structure that each item follows, which is going to be defined in Section 3.2.3.

As an example, the input dataset analyzed and exploited in this work to test the proposed methodology is the *status* of Barcelona's bike sharing stations over time, in particular between May and September 2008. The dataset was gathered by the authors of [16], and shared with professor Paolo Garza's research group for research and didactics purposes.

The *status* of each station is characterised by:

- A *timestamp* in the following format: "YYYY-MM-DD HH:MM:SS";
- The *stationID*;
- *used_slots* which is the number of bikes at the station, if less than or equal to a *threshold* the station is considered *empty*;
- *free_slots* which is the number of available slots at the station, if less than or equal to a *threshold* the station is considered *full*;

In addition, another related dataset is used throughout the work, which is the *stations* dataset containing additional information for each *stationID*, such as its *name*, *latitude* and *longitude*.

The preprocessing is composed of three ordered phases: filtering, alignment and sliding-window transformation.

3.2.1 Filtering

To begin with, only the logs associated to *full* or *empty* events are filtered in order to discard *normal* events which are way more frequent in the dataset as analyzed in Section 4.1. This is done in order to reduce the computational complexity of the frequent itemsets mining, as well as the association rules generation. An example of the input dataset with the events already extracted is presented in Table 3.1.

timestamp	**stationID**	**event**
2008-05-15 12:00:00	1	full
2008-05-15 12:01:00	3	full
2008-05-15 12:05:00	1	full
2008-05-15 12:05:00	2	empty
2008-05-15 12:10:00	1	full
2008-05-15 12:11:00	2	empty
2008-05-15 12:15:00	1	empty
2008-05-15 12:17:00	2	empty

Table 3.1: INPUT DATASET EXAMPLE

3.2.2 Alignment

It is then needed to align timestamps given a temporal *granularity* parameter of interest. Let's suppose we want to align events given a *granularity* of 5 minutes. Then, each timestamp has to be aligned to the lower minute 00, 05, 10, ..., 55. For example, given the data in Table 3.1, the alignment transformation is applied to get the resulting values in Table 3.2.

timestamp	stationID	event
2008-05-15 12:00:00	1	full
2008-05-15 12:00:00	3	full
2008-05-15 12:05:00	1	full
2008-05-15 12:05:00	2	empty
2008-05-15 12:10:00	1	full
2008-05-15 12:10:00	2	empty
2008-05-15 12:15:00	1	empty
2008-05-15 12:15:00	2	empty

Table 3.2: INPUT DATASET EXAMPLE AFTER ALIGNMENT

3.2.3 Sliding-window transformation

Following the alignment, the sequences are defined by applying a sliding-window of dimension *window size*. Each element of sequence follows the following format: "*delta_stationID_event*", where the *delta* is the relative temporal offset for the given window.

The resulting output of this phase is a transactional database where the concept of sequentiality is embedded in the items of the transaction. This is done to be able to plug-in the data to an algorithm such as FP-growth who needs the input data to be a transactional database.

Let's suppose we have a window dimension *window size* = 3 and the data in Table 3.2 as input, the resulting transactions by applying the sliding-window transformation are presented in Table 3.3.

transaction
0_1_full, 0_3_full, 1_1_full, 1_2_empty, 2_1_full, 2_2_empty
0_1_full, 0_2_empty, 1_1_full, 1_2_empty, 2_1_empty, 2_2_empty
0_1_full, 0_2_empty, 1_1_empty, 1_2_empty
0_1_empty, 0_2_empty

Table 3.3: INPUT DATASET EXAMPLE AFTER SLIDING-WINDOW TRANSFORMATION

3.3 Frequent itemsets mining

The pseudocode of the frequent itemsets mining algorithm is presented in Algorithm 1, Algorithm 2, Algorithm 3 and Algorithm 4.

Algorithm 1 is the main method of the frequent itemsets extraction and calls two functions: genFreqItems (Algorithm 2) and genFreqItemsets (Algorithm 3). Moreover, it creates the partitioner who is entitled to divide and distribute the data, this operation is called *Sharding* in Section 2.1.4.

genFreqItems counts the support of all items and filter those who have a greater support than *minSup* as well as ordering them by decreasing *support*, genFreqItems performs the step 2 and 3 of PFP, respectively called *Parallel Counting* and *Grouping Items*. The latter divides the items of *freqItemsCount* into Q groups that correspond to the partitions aforementioned.

genFreqItemsets is entitled to generate frequent itemsets by generating conditional transactions with function genCondTransactions (Algorithm 4) and by building the local FP-trees. This is the key step of the PFP algorithm. The transactions are converted into *group-dependent* transactions so that local FP-trees built from different *group-dependent* transactions are independent. In particular, in genCondTransactions for each transaction, the following step is performed: for each partition, if it appears in the transaction T_i, locate its right-most appearance, say L, and output a key-value pair $(key = partition, value = T_i[1], ..., T_i[L])$. Next, the AggregateByKey function in genFreqItemsets, for each shard it builds a local FP-tree and growth its conditional FP-trees recursively.

Furthermore, the implementation of the add method of the FP-Tree is modified by adding the spatial constraint. In particular, during the addition of a transaction to the tree, only items related to stations that are nearby each other within a *radius* parameter in meters are kept. This operation allows to prune the search space in order to improve the efficiency of the algorithm instead of applying a post-processing filter.

Algorithm 1 PFP-Growth to generate frequent itemsets

```
function run(RDD[Array[Item]] data)
    count ← data.count()                                  ▷ number of transactions
    minCount ← math.ceil(minSup * count)                  ▷ minimum support count
    partitioner ← new HashPartitioner(numParts)
    freqItemsCount ← genFreqItems(data, minCount)
    freqItemsets ← genFreqItemsets(data, minCount, freqItemsCount.map(_._1))
    itemSupport ← freqItemsCount.map{(item, cnt) ⇒ item → cnt/count}.toMap()
    return FPGrowthModel(freqItemsets, itemSupport)
end function
```

Algorithm 2 PFP-Growth genFreqItems function

```
function genFreqItems(RDD[Array[Item]] data, Long minCount)
    return data.map(v ⇒ (v, 1))
        .reduceByKey(partitioner, _ + _)
        .filter(_._2 >= minCount)
        .collect()
        .sortBy(−_._2)
end function
```

Algorithm 3 PFP-Growth genFreqItemsets function

```
function genFreqItemsets(RDD[Array[Item]] data, Long minCount,
Array[Item] freqItems)
    itemToRank ← freqItems.zipWithIndex().toMap()
    return data.flatMap{ transaction ⇒
        genCondTransactions(transaction, itemToRank)
        }.aggregateByKey(new FPTree[Int],
            (tree, transaction) ⇒ tree.add(transaction, 1),
            (tree1, tree2) ⇒ tree1.merge(tree2))
        .flatMap{ (part, tree) ⇒
            tree.extract(minCount)
        }.map { (ranks, count) ⇒
            new FreqItemset(ranks.map(i ⇒ freqItems(i)).toArray(), count)
        }
end function
```

Algorithm 4 PFP-Growth genCondTransactions function

```
function genCondTransactions(Array[Item] transaction,
Map[Item, Int] itemToRank)
    output ← Map.empty[Int, Array[Int]]
    filtered ← transaction.flatMap(itemToRank.get())
    Arrays.sort(filtered)
    n ← filtered.length
    i ← n − 1
    while i >= 0 do
        item ← filtered(i)
        part ← getPartition(item)
        if not output.contains(part) then
            output(part) ← filtered.slice(0, i + 1)
        end if
        i ← i − 1
    end while
    return output
end function
```

3.4 Association rules generation

The pseudocode of association rules generation is presented in Algorithm 5.

After generating the association rules we keep only the ones whose confidence is greater than a *minConf* parameter.

The PARTITION function returns two collections: one that satisfies the predicate function, the other that does not.

The **Rule** class is composed of five attributes:

- the *antecedent* as an Array[Item];
- the *consequent* as an Array[Item];
- *freqUnion*;
- *freqAntecedent*;
- *freqConsequent*.

The *confidence* of the rule is computed as $\frac{freqUnion}{freqAntecedent}$.

Algorithm 5 Association rules generation

```
function RUN(RDD[FreqItemset[Item]] freqItemsets, Map[Item, Double] itemSupport)
    //For candidate rule X ⇒ Y, generate (X, (Y, freq(XunionY)))
    candidates ← freqItemsets.FLATMAP{itemset ⇒
        items ← itemset.ITEMS()
        items.FLATMAP{item ⇒
            items.PARTITION(_ == item) match {
                (consequent, antecedent) ⇒
                    ((antecedent.TOSEQ(), (consequent.TOSEQ(), itemset.FREQ)))
            }
        }
    }
    //Join to get (X, ((Y, freq(X union Y)), freq(X))), generate rules, and filter by confidence
    return candidates.JOIN(freqItemsets.MAP(x ⇒ (x.ITEMS().TOSEQ(), x.FREQ())))
        .MAP{ (antecedent, ((consequent, freqUnion), freqAntecedent)) ⇒
            new Rule(antecedent.TOARRAY(),
                consequent.TOARRAY()
                freqUnion,
                freqAntecedent,
                itemSupport.GET(consequent.HEAD())   ▷ consequent is only one element
        }.FILTER(_.CONFIDENCE() >= minConf)
end function
```

3.4.1 Post processing filter

After having generated the association rules it is needed to apply a post processing filter to keep the ones:

- whose *delta* in the consequent is after all *deltas* in the antecedent, in order to have rules that predict a value in the future;
- who have at least one delta set to 0, in order to discard shifted duplicates.

Given a *granularity* of 30 minutes, an example of valid association rule is the following:

$$0\_1\_empty, 1\_2\_full \Rightarrow 2\_3\_empty\ [Support = 10\%, Confidence = 75\%] \tag{3.1}$$

Which means: whenever station 1 is empty, if after 30 minutes station 2 is full, then there is a 75% probability that station 3 will be empty after another 30 minutes, and this happens 10% of times in the dataset.

3.5 Prediction model training

The prediction model is based on applying associative classification on the input sequences. In particular, the input transactional dataset is partitioned into training set and test set as follows:

- **Training set**: $\frac{2}{3}$ of the input transactional dataset on which the frequent itemsets mining and the association rule generation is performed;
- **Test set**: the remaining $\frac{1}{3}$ of the input transactional dataset on which the associative classifier is tested.

The associative classifier works as follows:

- First, the association rules extracted for the training set are ordered by in sequence:
  - Decreasing confidence;
  - Decreasing support;
  - Rule length;
  - Lexicographical order;
- Then, given for example a $window\ size = 3$, we iterate over the transactions of the test set and we check for each station at $delta = 2$, following the order defined previously, what is the first rule that can be applied (in terms of matching the elements of the *antecedent* and matching the station for the *consequent*) and if there is a match, we emit a prediction, which is the *consequent* of the rule. If no rule can be applied, the classifier predicts for the current station the *normal* event. Next, we check if the prediction is correct or not, producing a confusion matrix for each station, a total confusion matrix collecting all the predictions, as well as computing the average precision, average recall and accuracy metrics along with precision and recall for each class. An overview of the several types of classifiers tested is presented in Section 4.3.

Transaction
0_1_full, 0_3_full, 1_1_full, 1_2_empty, 2_1_full, 2_2_empty
0_1_full, 0_2_empty, 1_1_full, 1_2_empty, 2_1_empty, 2_2_empty
...

Table 3.4: TEST SET TRANSACTIONS FOR CLASSIFICATION EXAMPLE

3.5.1 Example

To better understand how the associative classifier works, an example is presented.

Ordered association rules
$0\_3\_full, 1\_2\_empty \Rightarrow 2\_1\_full$
$0\_1\_full, 1\_1\_full \Rightarrow 2\_2\_full$
$0\_1\_empty \Rightarrow 2\_3\_empty$

Table 3.5: ORDERED ASSOCIATION RULES FOR CLASSIFICATION EXAMPLE

Given the transactions in Table 3.4 and the ordered association rules in Table 3.5 we iterate over the elements of the first transaction until we find an item with $delta = 2$.

The first item is 2_1_full then, we go through the ordered association rules and we look for the first one who has in the consequent 2_1_X (X means don't care). The first rule $0\_3\_full, 1\_2\_empty \Rightarrow 2\_1\_full$ matches the consequent we are interested in. Afterwards we check if the elements in the antecedent of the rule are present in the transaction, then we have a match for the rule. Therefore we predict *full* for station 1 at $delta = 2$ and it is a correct prediction given that the item in the transaction is 2_1_full.

On the other hand, an example of wrong prediction is the following: the second item is 2_2_empty, then we scan through the association rules looking for the first one who has in the consequent 2_2_X. The second rule $0\_1\_full, 1\_1\_full \Rightarrow 2\_2\_full$ matches the consequent we are interested in. Then we check if the elements in the antecedent are present in the transaction, consequently we have a match for the rule. Hence, we predict *full* for station 2 at $delta = 2$ and it is a wrong prediction given that the item in the transaction is 2_2_empty.

Moreover, when a station like station 3 as in the first transaction does not appear with $delta = 2$ we assume that the station is *normal* (2_3_normal). Then we look for the first rule who has in the consequent 2_3_X. The third rule $0\_1\_empty \Rightarrow 2\_3\_empty$ matches the consequent we are interested in. Then we check if the elements in the antecedent are present in the transaction, then we don't have a match for the rule. Since no rules can be applied, we predict *normal* for station 3 at $delta = 2$ and it is a correct prediction given that the item in the transaction is 2_3_normal.

Chapter 4

Experimental results

4.1 Dataset analysis

The dataset that will be tested throughout the work contains the historical information about Barcelona's bike sharing system between May and September 2008, where the number of stations involved is 284. Figure 4.1 shows the location of the stations.

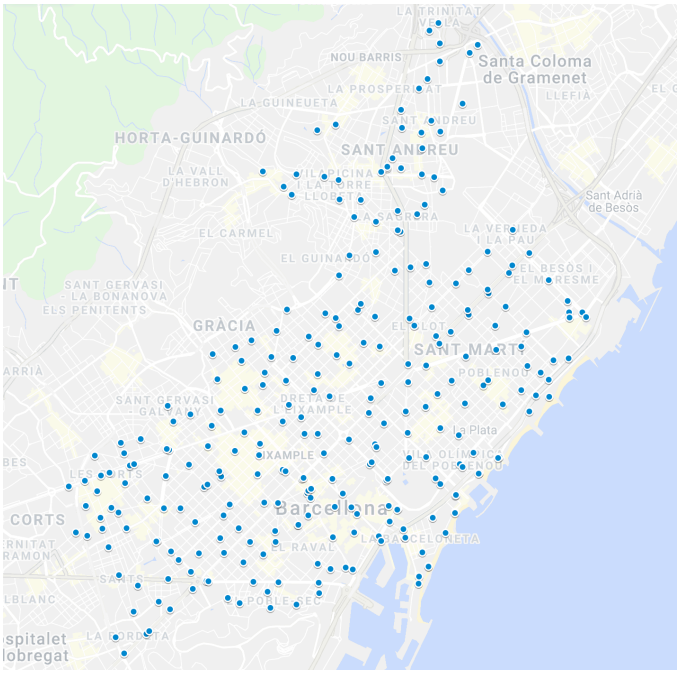
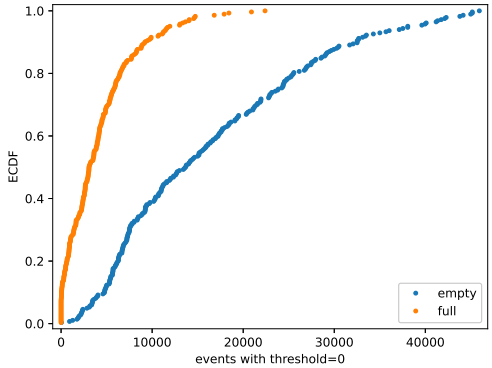


Figure 4.1: Locations of bike sharing stations

The first analysis is conducted on the number of events per station, it is then produced the Empirical Cumulative Distribution Function (ECDF) varying the *threshold* parameter from Figure 4.2 to Figure 4.9. The ECDF allows to plot a feature of the data in order and observe the feature as distributed across the dataset. It can be read as follows: for

example given Figure 4.2, what is the percentage of stations that has less than 10000 full events? Look for 10000 on the x-axis and then move vertically until hitting the curve. The answer is: nearly 90% of stations has less than 10000 full events.



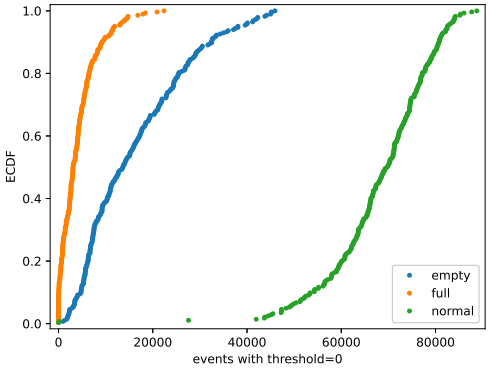
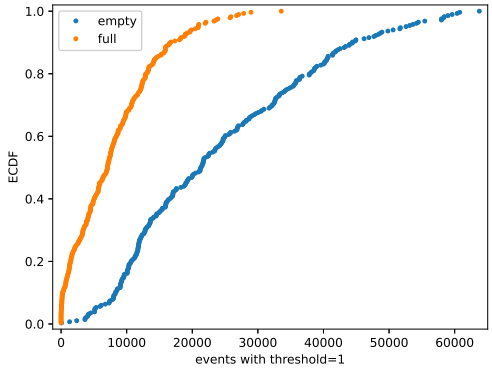


Figure 4.2: Events per station ECDF thr = 0Figure 4.3: Events per station ECDF thr = 0



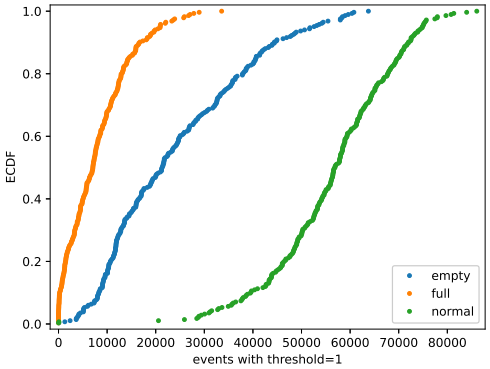


Figure 4.4: Events per station ECDF thr = 1Figure 4.5: Events per station ECDF thr = 1

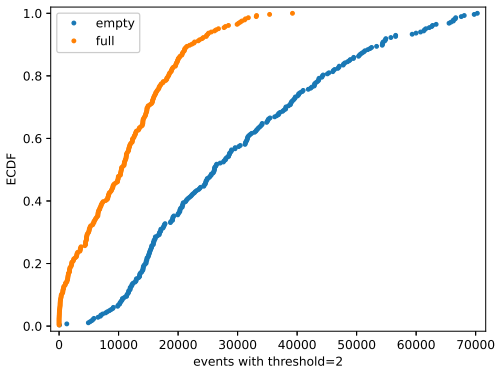


Figure 4.6: Events per station ECDF thr = 2

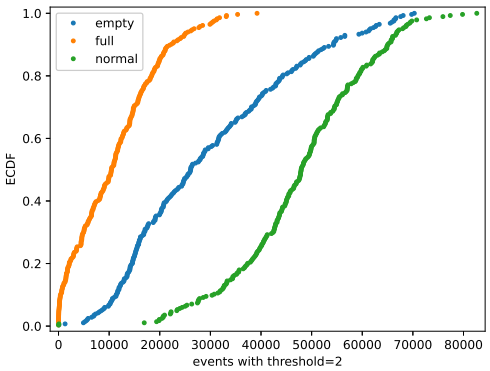


Figure 4.7: Events per station ECDF thr = 2

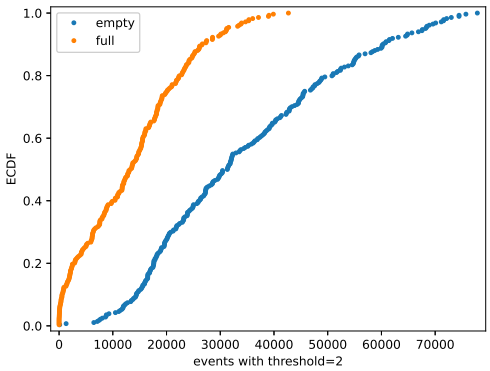


Figure 4.8: Events per station ECDF thr = 3

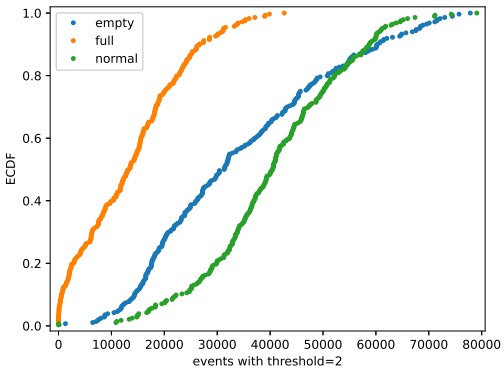


Figure 4.9: Events per station ECDF thr = 3

We can see that the most rare events are the *full* ones, as opposed to the *normal* events which are the most frequent. In addition, we verify that as the threshold increases the number of *full* and *empty* events per station increases too.

4.2 Parameters effect on association rule generation

In this section we are going to analyze the effects of model parameters on the extraction of association rules in terms of:

- Execution time in seconds
- Number of frequent itemsets
- Number of association rules
- Distribution of rules confidence

The default parameters are:

- $threshold = 0$;
- $granularity = 5\,minutes$;
- $window\,size = 3$;
- $minSup = 0.2$;
- $radius = 500m$;
- $minConf = 0.8$.

4.2.1 Testing environment

The parallel code used to perform frequent itemsets mining and association rule generation was run on the machines of the *SmartData@Polito* cluster located at the *Politecnico di Torino*, Italy. A Yarn queue was dedicated to the execution of the job, in particular for each job 4 executors are instantiated, each equipped with 4GB of main memory.

4.2.2 Radius

The choice of having *radius* values between 500m, 1km and 1.5km is due to the fact that if a user, for example, wants to reach a nearby station, those values are reasonable for a walk route. As shown in Figure 4.10, Figure 4.11 and Figure 4.12: execution time, the number of itemsets and the number of rules increases as the *radius* rises. Figure 4.13 shows the distribution of rules confidence given different *radius*.

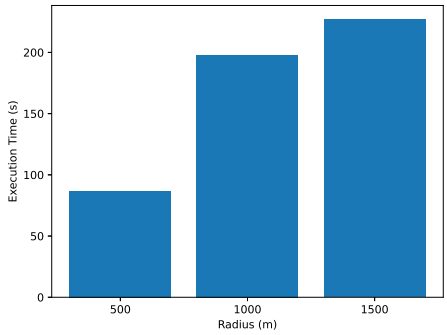


Figure 4.10: Execution time by *radius*

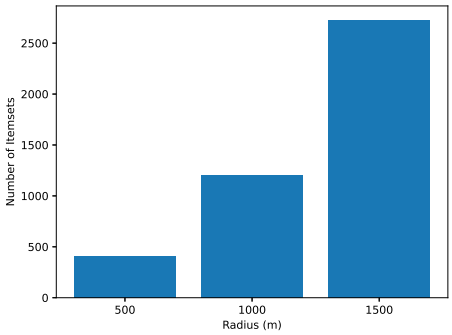


Figure 4.11: Number of itemsets by *radius*

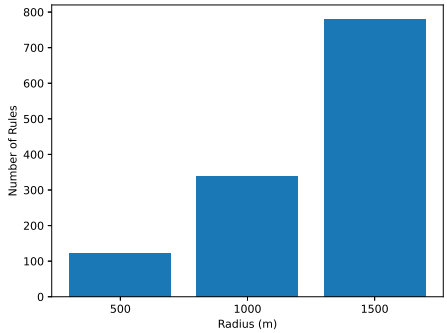


Figure 4.12: Number of rules by *radius*

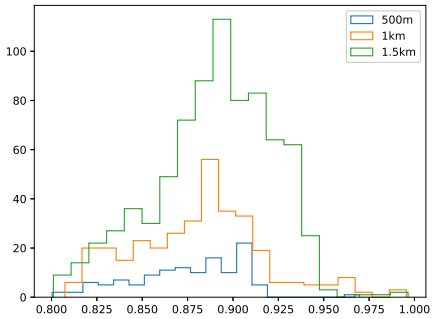


Figure 4.13: Distribution of rules confidence

4.2.3 MinSup

As shown in Figure 4.14, Figure 4.15 and Figure 4.16: the execution time, the number of itemsets and the number of rules decreases as the *minSup* value increases. Figure 4.17 shows the distribution of rules confidence given different *minSup*.

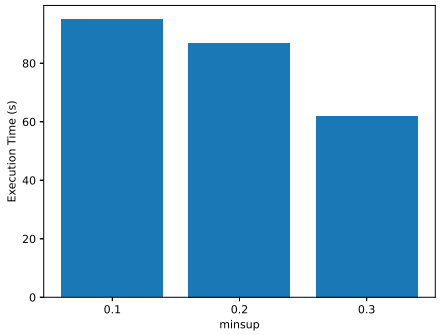


Figure 4.14: Execution time by *minSup*

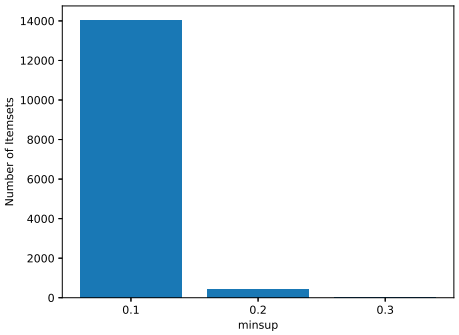


Figure 4.15: Number of itemsets by *minSup*

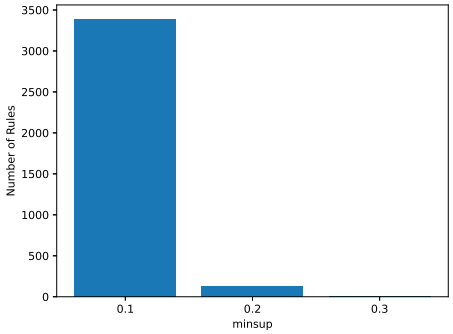


Figure 4.16: Number of rules by *minSup*

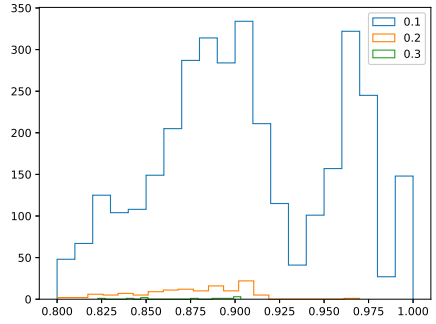


Figure 4.17: Distribution of rules confidence

4.2.4 Window size

The choice of having a *window size* between 3, 4 and 5 is due to the fact that having a window composed by less than 3 items is not interesting for the analysis, whereas, a window composed by more than 5 items is computationally expensive. As shown in Figure 4.18, Figure 4.19 and Figure 4.20: the execution time, the number of itemsets and the number of rules increases as the *window size* rises. Figure 4.21 shows the distribution of rules confidence given a different *window size*.

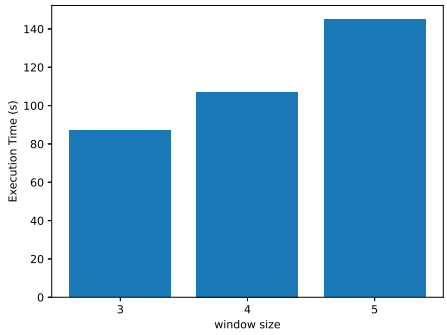


Figure 4.18: Execution time by *window*

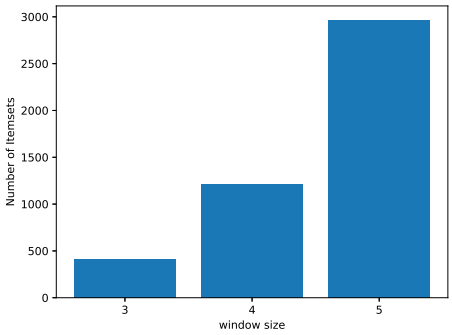


Figure 4.19: Number of itemsets by *window*

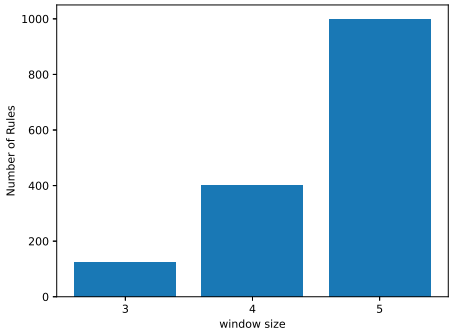


Figure 4.20: Number of rules by *window*

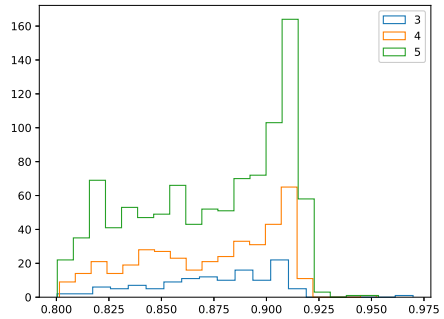


Figure 4.21: Distribution of rules confidence

4.2.5 Granularity

The choice of having a *granularity* between 5, 15, 30 and 60 minutes is due to the fact that window dimensions are reasonable since having values less than 5 minutes and more than 60 minutes would not be interesting for the analysis. As shown in Figure 4.22 the execution time decreases as the granularity increases, because of the decrease of the number of input transactions. On the other hand, as shown in Figure 4.23 and Figure 4.24, both the number of itemsets and the number of rules increases as the granularity rises. Figure 4.25 shows the distribution of rules confidence given different *granularity*.

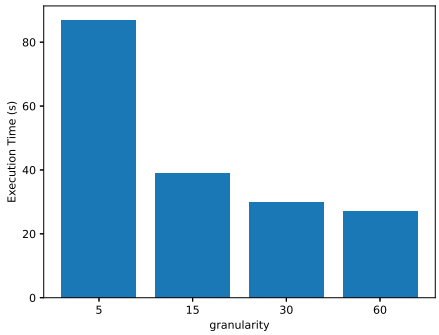


Figure 4.22: Execution time by *gran*

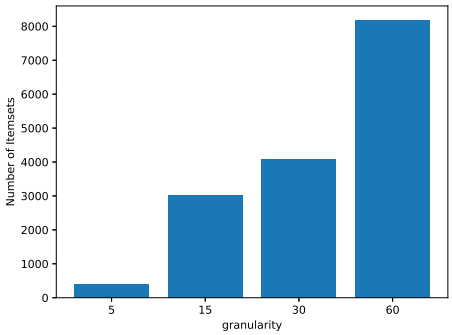


Figure 4.23: Number of itemsets by *gran*

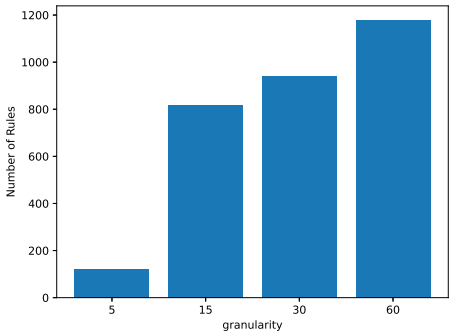


Figure 4.24: Number of rules by *gran*

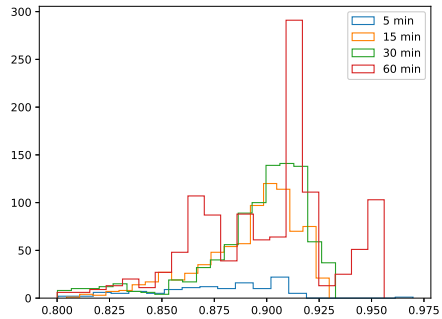


Figure 4.25: Distribution of rules confidence

4.3 Prediction model training

4.3.1 Testing environment

The sequential code for the prediction model training was run on a macOS Mojave 10.14.6 PC equipped with an Intel Core i7 (2.6 GHz) CPU and 16 GB of RAM. The version of the Python interpreter is the 3.9 and we used the `scikit-learn` library [17] to train and test several competitor algorithms, as well as the `matplotlib` library [18] to plot the graphs.

4.3.2 Categories of event

A station can be characterised by two types of critical events: *empty* when the value of *used_slots* is lower than or equal to the *threshold* parameter or *full* when the value of *free_slots* is lower than or equal to the *threshold* parameter. In addition, we present two definitions of criticalities:

- **All critical** events, when we consider all the logs that contain an empty or full station;

- **Becoming critical** events, when we consider only the events located at the beginning of a series of critical events.

Given a $threshold = 0$ and a *granularity* of 2 minutes, the difference between the two definitions is presented in Table 4.1.

timestamp	stationID	used_slots	free_slots	all critical	becoming critical
2008-06-01 11:40:00	1	1	18	/	/
2008-06-01 11:42:00	1	0	19	empty	empty
2008-06-01 11:44:00	1	0	19	empty	/
2008-06-01 11:46:00	1	0	19	empty	/
2008-06-01 11:46:00	1	1	18	/	/

Table 4.1: SAMPLE OF RECORDS AND EXTRACTION OF EVENTS

4.3.3 Timeslots

Some of the following experimental trials will be enriched with the concept of timeslots. In particular, we divide the hours of the day into slots of 4-hours and we add this feature as an item of each transaction. Therefore, some rules can also include a timeslot in their antecedent. Given a *granularity* of 30 minutes we can have a rule as follows:

$$0\_61\_full, 1\_70\_full, ts\_-1\_20-24 \Rightarrow 2\_61\_full\ [Support = 1\%, Confidence = 98\%] \tag{4.1}$$

Which means: whenever station 61 is full, if after 30 minutes station 70 will be full and if we are between hour 20 and 24 of the day, in another 30 minutes time there is a 98% probability that station 61 will be full, and this happens 1% of times in the dataset.

4.3.4 Tested classifiers

In the following subsections several types of associative classifiers are presented. The result we aim to obtain is to achieve a high precision for the critical classes *empty* and *full*, in particular for class *full* since it is the most rare event. The following experiments are characterized by different configurations. For instance, by using the *all critical* or the *becoming critical* semantic defined in Section 4.3.2, moreover by focusing on binary classifications like *full/not full* or *empty/not empty*. In addition, competitor classifiers metrics are presented and compared to the associative classifiers. Eventually, multiple-layers classifiers are described such as the double binary classifiers as well as the mixed approach ones.

From Section 4.3.4 to Section 4.3.4 the default parameters are:

- $threshold = 0$;
- $granularity = 30\, minutes$;
- $window\, size = 3$;

- $minSup$ is variable, specified in the x-axis
- $radius = 500m$;
- $minConf = 0.98$;
- $ruleLen = 0$, it represents the minimum rule length.

AEAFT classifier

The AEAFT (All Empty All Full Timeslots) classifier is characterized as follows:

- In the generation of association rules it considers all *empty* and all *full* events, therefore rules can be composed of elements characterised by the *empty* event as well as the *full* event;
- If no rule is matched it emits a *normal* prediction;
- Timeslots are included.

This experiment is being done in order to extract rules that contain both *empty* and *full* events. Its metrics are presented in Figure 4.26.

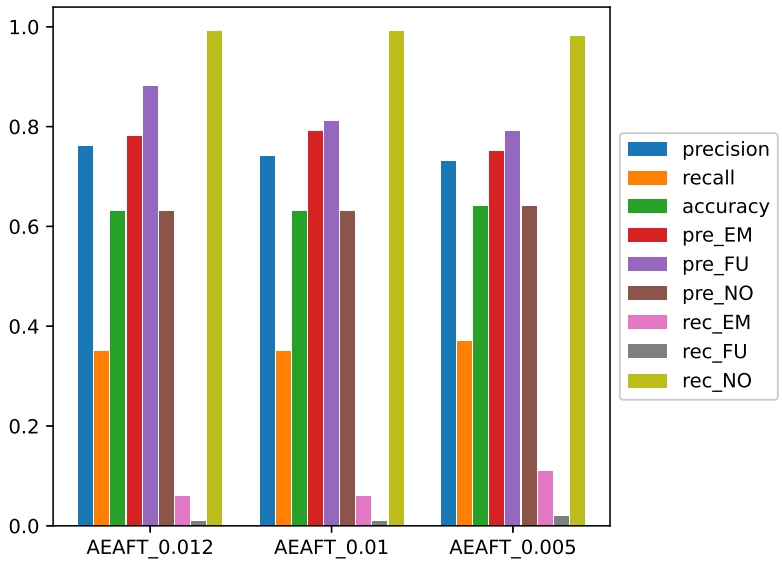


Figure 4.26: AEAFT classifier metrics by AEAFT_minSup

We can observe that the AEAFT classifier achieves a very high precision for the critical class *full* (values higher than 80%), on the other hand precision is only around 60% for the *normal* class. Moreover, a weak point is the very low values for recalls, except for the *normal* class. On balance, we can see that the precision decreases as the *minSup* lowers whereas, recalls slightly increase.

AFT,AF classifiers

The AFT (All Full Timeslots) classifier is characterized as follows:

- In the generation of association rules we consider only *full* events, therefore it can be considered as a binary classifier (*full*/*not full*);
- If no rule is matched we emit a *not full* prediction;
- Timeslots are included.

The AF classifier is the same as AFT except that it does not include timeslots. These experiments are being done in order to take into account only *full* events and see how the classifiers perform in such conditions. Metrics for AFT and AF are presented in Figure 4.27.

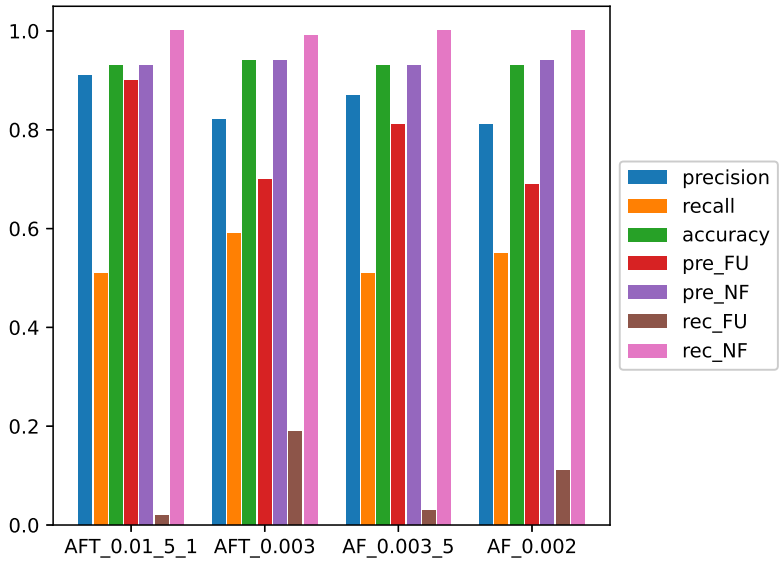


Figure 4.27: AFT, AF classifiers metrics by AFX_minSup_ruleLen_minConf

We can point out that accuracy is very high for both models. In addition, precision decreases, while recall slightly increases as the *minSup* decreases. In particular, we can see how the presence of timeslots in AFT is crucial since it increases the precision.

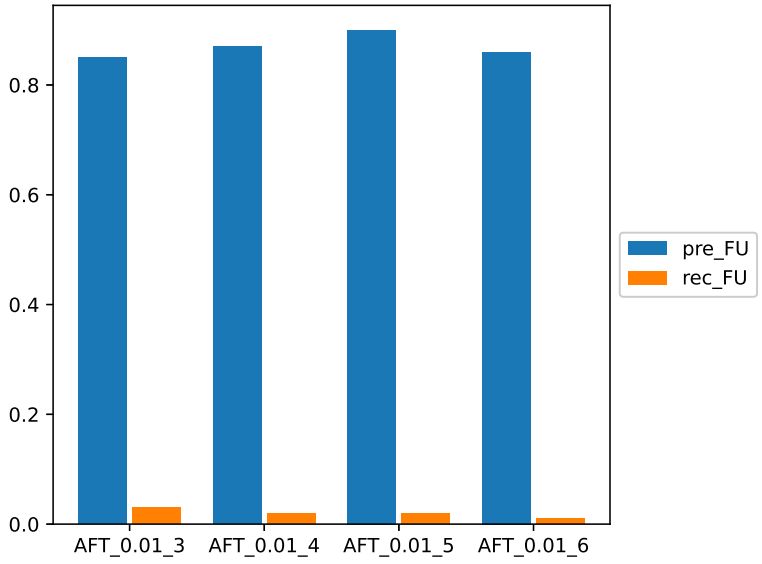


Figure 4.28: AFT classifier metrics by AFT_minSup_ruleLen

In Figure 4.28 we can observe how as the *ruleLen* increases the precision for the class *full* grows, until 5, then decreases. Moreover, the recall for the class *full* gradually decreases as the *ruleLen* increases.

AET,AE classifiers

The AET (All Empty Timeslots) classifier is characterized as follows:

- In the generation of association rules we consider only *empty* events, therefore it can be considered as a binary classifier (*empty*/*not empty*);
- If no rule is matched we emit a *not empty* prediction;
- Timeslots are included.

The AE classifier is the same as AET except that it does not include timeslots. These experiments are being carried out in order to take into account only *empty* events and see how the classifiers perform in such conditions. Metrics for AET and AE are presented in Figure 4.29.

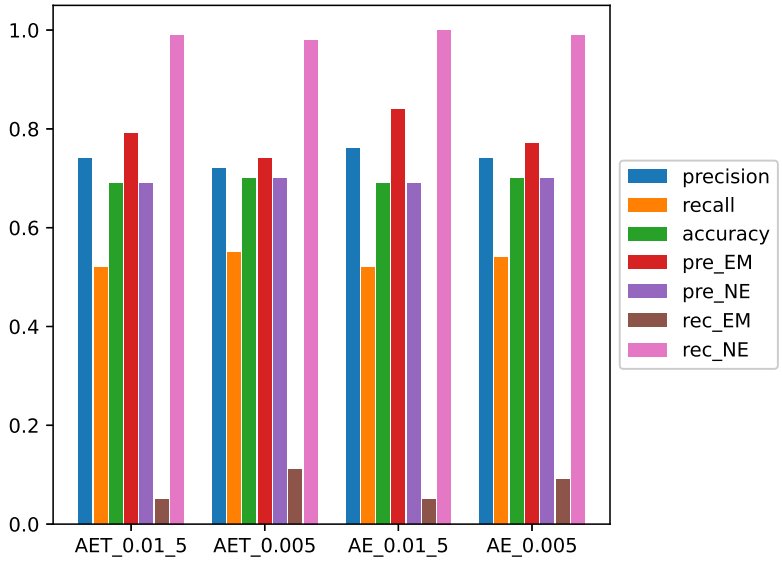


Figure 4.29: AET, AE classifiers metrics by AEX_minSup_ruleLen

We can see how as the *minSup* lowers precision for class *empty* decreases. In addition, as opposed to the previous classifier AFT, the presence of timeslots in AET decreases the precision for the class *empty* with respect to AE.

BFT,BF classifiers

The BFT (Becoming Full Timeslots) classifier is characterized as follows:

- In the generation of association rules we consider only *becoming full* critical events, therefore it can be considered as a binary classifier (*full*/*not full*);
- If no rule is matched we emit a *not full* prediction;
- Timeslots are included.

The BF classifier is the same as BFT except that it does not include timeslots. These experiments are being done in order to take into account the *becoming full* events and see how the classifiers perform in such conditions. Metrics for BF and BFT are presented in Figure 4.30.

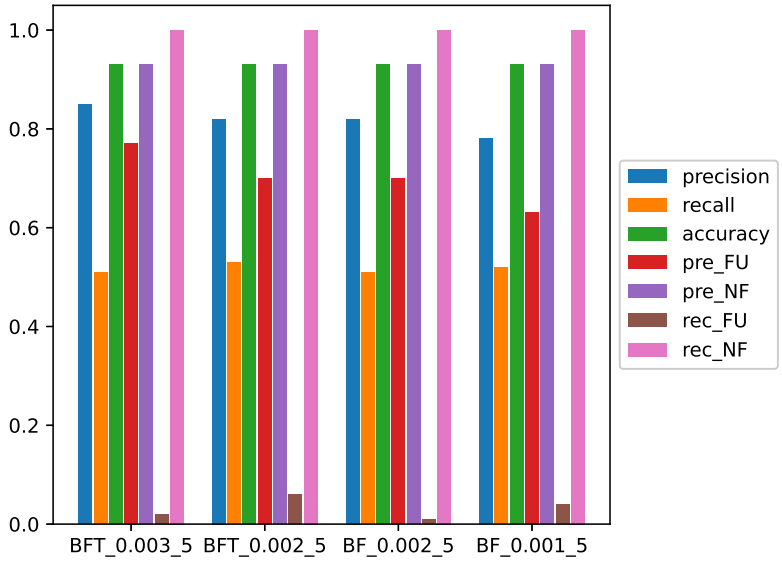


Figure 4.30: BFT, BF classifiers metrics by BFX_minSup_ruleLen

We can comment that the presence of timeslots increases the values of precision for the class *full*, even if, above all the values of precision are low (less than 80%). Besides, we achieve a very high value of accuracy with both classifiers and we can notice that the precision decreases as the *minSup* reduces, whilst recall marginally increases.

BET,BE classifiers

The BET (Becoming Empty Timeslots) classifier is characterized as follows:

- In the generation of association rules we consider only *becoming empty* critical events, therefore it can be considered as a binary classifier (*empty*/*not empty*);
- If no rule is matched we emit a *not empty* prediction;
- Timeslots are included.

The BE classifier is the same as BET except that it does not include timeslots. These experiments are being carried out in order to take into account the *becoming empty* events and see how the classifiers perform in such conditions. Metrics for BET and BE are presented in Figure 4.31.

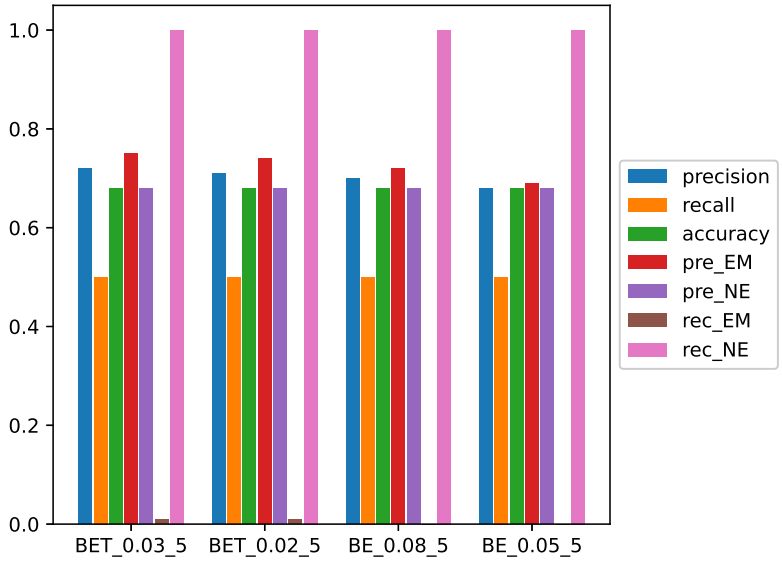


Figure 4.31: BET, BE classifiers metrics by BEX_minSup_ruleLen

As in the previous classifiers, we can spot that the presence of timeslots increases the values of precision for class *empty*. Furthermore, recall values for class *empty* are very low (1% or less).

Double binary classifiers

The double binary classifiers are characterized by a 2-layer architecture:

- In the first layer we consider rules from a binary classifier (*full*/*not full*);
- If no rule is applied we consider rules from a binary classifier (*empty*/*not empty*);
- If no rule is matched we emit a *normal* prediction for the interested station.

These experiments are being conducted in order to analyze if using a different architecture based on different layers could improve the metrics. Metrics for the double binary classifiers are shown in Figure 4.32.

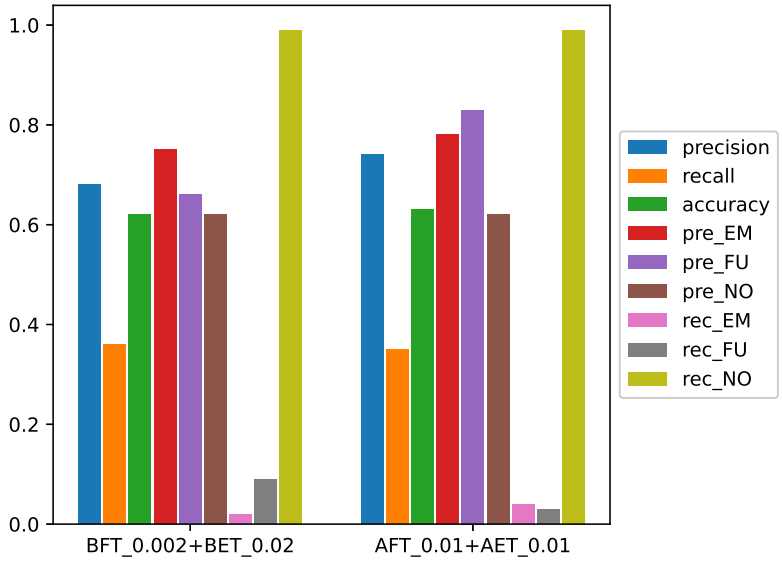


Figure 4.32: Double binary classifiers metrics by XXX_minSup+XXX_minSup

We can point out that the double binary classifier AFT+AET achieves higher values for critical classes (*full* and *empty*) precision with respect to BFT+BET. In addition, AFT+AET reaches marginally higher value for *empty* recall and lower value for *full* recall.

Competitor classifiers

Competitor classifiers have the following characteristics:

- One model for each station is generated;
- They are trained with the same input data (transactions) as the one used for training our models;
- The training data consists of the status of all bike sharing stations at $delta = 0$ and $delta = 1$ for each transaction.

After performing GridSearch to tune the hyperparameters for the competitors classifiers, their metrics are shown in Figure 4.33.

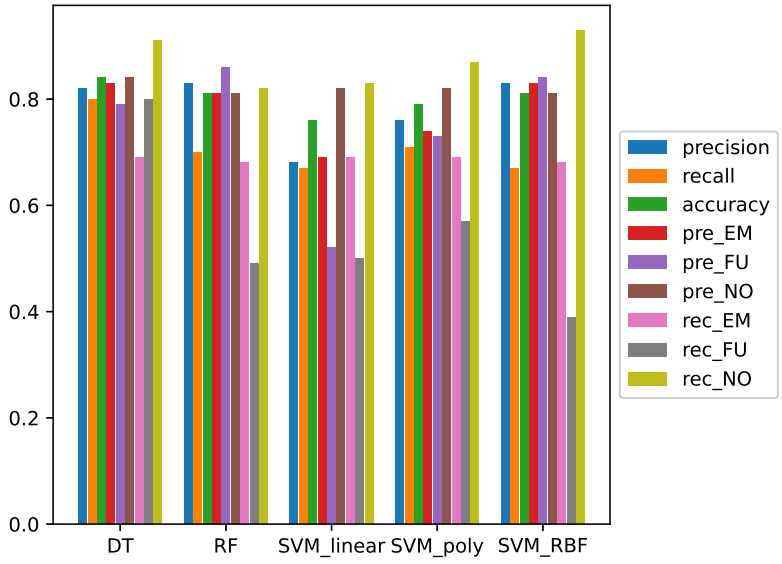


Figure 4.33: Competitor classifiers metrics

We can observe the Decision Tree (DT) as the model that better performs, in terms of tradeoff between precision and recall for all classes. Whereas, Random Forest (RF) obtains higher precision for *full* class despite a low value of its recall. The SVM that better performs is the one with the RBF kernel, it reaches similar values of classes precision with respect to the Decision Tree at the expense of lower value of recall for *full* class.

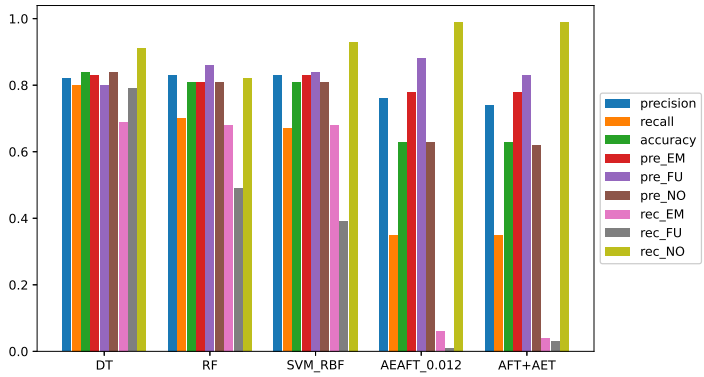


Figure 4.34: Comparison with competitor classifiers metrics

When it comes to compare our methodology to the mentioned competitor algorithms we can notice in Figure 4.34 as we manage to achieve, for the AEAFT classifier, a higher

precision for the *full* class at the expense of very low values for critical classes (*empty* and *full*) recalls, as well as a lower precision for the *normal* class. On balance, our methodology does not outperform competitor algorithms if we take into account the tradeoff between precision and recall.

Mixed approach classifiers

The mixed approach classifier DT+AFT is characterized by a 2-layer architecture:

- In the first layer we apply a binary Decision Tree classifier (*normal*/*not normal*);
- If the prediction is *not normal*, in the second layer we consider rules from a binary classifier (*full*/*not full*);
- If no rule is matched we emit an *empty* prediction for the interested station.

These experiments are being conducted in order to analyze if using a mixed architecture based on different classifiers could improve the metrics. In particular, the target is to higher the recall for the critical classes *empty* and *full*. The metrics for the DT+AFT classifier are reported in Figure 4.35.

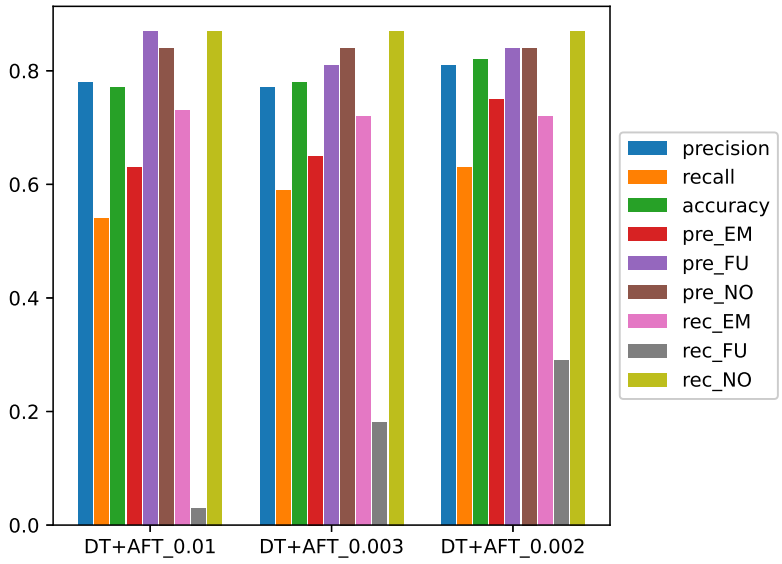


Figure 4.35: Mixed approach DT+AFT classifiers metrics by DT+AFT_minSup

With respect to previous classifiers, the DT+AFT classifier manages to higher the recall for the *empty* class despite a lower precision. Moreover, as the *minSup* decreases the recall for class *full* increases, as well as the precision for the *empty* class that increases.

Furthermore, if we want to consider only the most critical and rare class, which is the *full* one (as analysed in Section Figure 4.1), we can design a AFT+DT binary classifier that

classifies between *full* and *not full*. The AFT+DT classifier is characterized by a 2-layer architecture:

- In the first layer we consider the association rules from the AFT binary classifier (*full*/*not full*);
- If no rule is matched, in the second layer we apply a binary Decision Tree classifier (*full*/*not full*).

The target of this classifier is to increase the recall for the *full* class, as well as achieving a very high precision for the first layer. Metrics for the AFT+DT classifier are compared to a binary Decision Tree classifier (*full*/*not full*) in Figure 4.36.

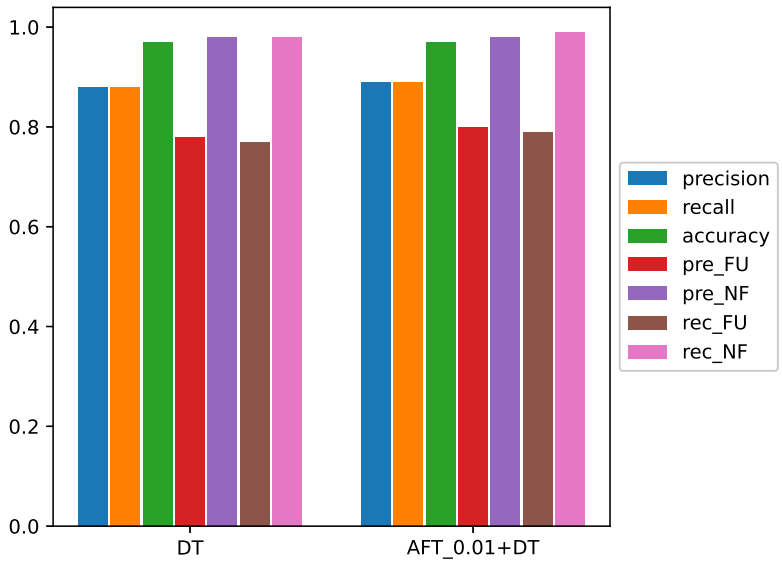


Figure 4.36: Mixed approach AFT+DT classifiers metrics by AFT_minSup+DT

We can mention that both classifiers reach the same values for precisions: 78% for class *full* and 98% for class *not full*. Additionally, the AFT+DT classifier achieves a rather higher value of recall for class *full*. On balance both classifiers are comparable in terms of metrics, but the advantage is that we achieve a very high precision in the first layer (90%), providing also explainability for those cases.

Lastly, Figure 4.37 shows a comparison between all associative classifiers tested, focusing only on precision and recall for class *full*, since we would be interested in achieving high precision for the most rare event.

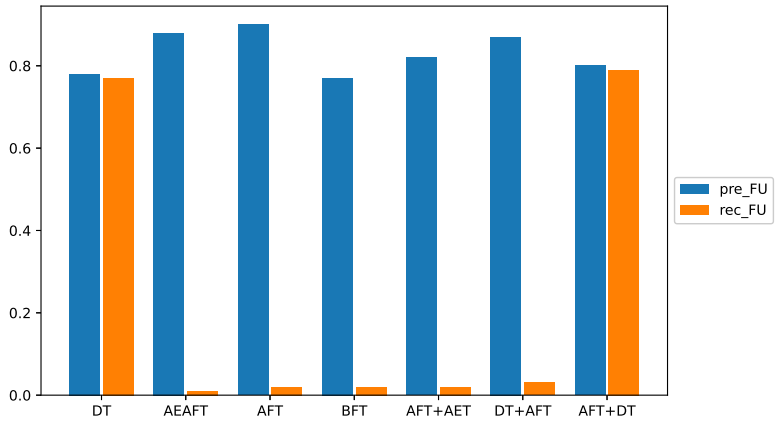


Figure 4.37: Associative classifiers comparison

We can point out that the classifier that achieves the highest precision is the AFT, at the expense of a very low recall (only 2%). The classifiers that outperform the Decision Tree in terms of precision are: AEAFT, AFT, AFT+AET and DT+AFT, despite a significant lower recall. The mixed approach classifier AFT+DT achieves similar value of precision with respect to the Decision Tree, whereas reaching slightly higher value of recall.

Chapter 5

Conclusions

The target of this work was to introduce a methodology to perform associative classification on spatio-temporal sequences. We believe that such methodology could be applied in a broad range of applications, given that the dataset follows the concept of states or events associated to space and time.

Competitor algorithms such as Decision Trees try to choose paths in the tree in order to maximize the result quality for all classes. Whereas, each association rule is independently extracted from the others, and it is kept if confidence is greater than a *minConf* threshold. Moreover, in each association rule we have conditions on a certain set of attributes that can be different to the set of attributes of a different rule. In addition, association rules take into account correlation among the several events, while the decision tree is built step by step impacting on what can be extracted. Since the Decision Tree considers one attribute at a time, some rules are discarded during the creation of the tree. On the other hand, in association rules, we extract all the possible frequent combinations leading to poor metrics in terms of tradeoff between precision and recall. Furthermore, one advantage of using association rules is the high explainability with respect to the competitor algorithms.

On balance, the advantage of performing associative classification is achieving very high precisions for the critical and most rare class *full*, at the expense of a very low recall. In order to improve the recall, we designed a 2-layer classifier, where in the first layer we use association rules, which provide explainability and are characterized by a very high precision (90%). Then, we pass the cases not covered by the association rules to a second layer where we use a Decision Tree, which increases the recall of the whole architecture.

Bibliography

[1] T. Imielinski, Arun Swami, and Rajat Agrawal. Mining association rules between sets of items in large databases. *ACM SIGMOD*, pages 207–216, 01 1993.

[2] R. Agrawal. Fast algorithms for mining association rules. *the Proc. of 20th Int.Conf. on Very Large Databases (VLDB), Santiago de Chile, Chile*, 01 1994.

[3] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. volume 29, pages 1–12, 06 2000. doi: 10.1145/342009.335372.

[4] Haoyuan Li, Yi Wang, Dong Zhang, Ming Zhang, and Edward Chang. Pfp: parallel fp-growth for query recommendation. pages 107–114, 01 2008. doi: 10.1145/1454008.1454027.

[5] Ramakrishnan Srikant and Rakesh Agrawal. Mining sequential patterns: Generalizations and performance improvements. *EDBT, Lecture notes in computer science. vol. 1057*, 1057:1–17, 03 1996. doi: 10.1007/BFb0014140.

[6] Pokou Jean, Philippe Fournier Viger, and Chadia Moghrabi. Authorship attribution using variable length part-of-speech patterns. pages 354–361, 01 2016. doi: 10.5220/0005710103540361.

[7] Philippe Fournier Viger, Ted Gueniche, and Vincent Tseng. Using partially-ordered sequential rules to generate more accurate sequence prediction. pages 431–442, 12 2012. ISBN 978-3-642-35526-4. doi: 10.1007/978-3-642-35527-1_36.

[8] Philippe Fournier Viger, Roger Nkambou, and Engelbert Mephu Nguifo. A knowledge discovery framework for learning task models from user interactions in intelligent tutoring systems. pages 765–778, 01 2008.

[9] Jianyong Wang, Jiawei Han, and Chun Li. Frequent closed sequence mining without candidate maintenance. *Knowledge and Data Engineering, IEEE Transactions on*, 19: 1042–1056, 09 2007. doi: 10.1109/TKDE.2007.1043.

[10] Mohammed Zaki. Zaki, m.j.: Spade: An efficient algorithm for mining frequent sequences. machine learning 42(1), 31-60. *Machine Learning*, 42:31–60, 01 2001. doi: 10.1023/A:1007652502315.

[11] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Jianyong Wang, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. Mining sequential patterns by pattern-growth: The prefixspan approach. *Knowledge and Data Engineering, IEEE Transactions on*, 16:1424– 1440, 12 2004. doi: 10.1109/TKDE.2004.77.

[12] Jay Ayres, Jason Flannick, Johannes Gehrke, and Tomi Yiu. Sequential pattern mining using a bitmap representation. pages 429–435, 01 2002. doi: 10.1145/775107.775109.

[13] Zhenglu Yang and M. Kitsuregawa. Lapin-spam: An improved algorithm for mining sequential pattern. pages 1222– 1222, 05 2005. ISBN 0-7695-2657-8. doi: 10.1109/ICDE.2005.235.

[14] Philippe Fournier Viger, Antonio Gomariz, Manuel Campos, and Rincy Thomas. Fast vertical mining of sequential patterns using co-occurrence information. pages 40–52, 05 2014. ISBN 978-3-319-06607-3. doi: 10.1007/978-3-319-06608-0_4.

[15] Bing Liu, Wynne Hsu, and Yiming Ma. Integrating classification and association rule mining. In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*, KDD'98, page 8086. AAAI Press, 1998.

[16] Andreas Kaltenbrunner, Rodrigo Meza, Jens Grivolla, Joan Codina, and Rafael Banchs. Urban cycles and mobility patterns: Exploring and predicting trends in a bicycle-based public transport system. *Pervasive and Mobile Computing*, 6:455–466, 08 2010. doi: 10.1016/j.pmcj.2010.07.002.

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

[18] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi: 10.1109/MCSE.2007.55.