UNIVERSITÀ DI PISA
Scuola di Dottorato in Ingegneria "Leonardo da Vinci"

Corso di Dottorato di Ricerca in Ingegneria dell'Informazione

Tesi di Dottorato di Ricerca

Classification Algorithms for Big Data over Distributed Processing Frameworks

Armando Segatori

Anno 2016

UNIVERSITÀ DI PISA

Scuola di Dottorato in Ingegneria "Leonardo da Vinci"

Corso di Dottorato di Ricerca in
Ingegneria dell'Informazione

Tesi di Dottorato di Ricerca

Classification Algorithms for Big Data over Distributed Processing Frameworks

Autore:

Armando Segatori ______________________

Relatori:

Prof. Francesco Marcelloni ______________________

Ing. Alessio Bechini ______________________

Prof. Pietro Ducange ______________________

Anno 2016

Sommario

I problemi di classificazione sono stati a lungo studiati nel contesto del *data mining* e, negli ultimi decenni, sono stati sviluppati diversi approcci in grado di affrontare tali problemi. Tra questi, gli approcci basati sulla classificazione associativa e gli alberi di decisione si sono dimostrati molto efficaci e sono stati utilizzati con successo in diversi domini applicativi. Inoltre, alcuni di questi approcci hanno integrato la teoria degli insiemi fuzzy con l'obiettivo di generare classificatori capaci di tollerare dati incerti e rumorosi. Sfortunatamente, la maggior parte degli approcci proposti fino ad ora sono stati progettati per massimizzare l'accuratezza, trascurando spesso la complessità sia in termini di memoria che tempi di esecuzione. Cosí questi approcci non sono generalmente in grado di gestire in modo adeguato i cosiddetti "big data".

Il lavoro presentato in questa tesi di dottorato propone differenti soluzioni in ambiente distribuito per costruire modelli di classificazione accurati e interpretabili per "big data". In particolare, sono stati considerati approcci basati sulla classificazione associativa e su alberi di decisione, integrando le soluzioni proposte con la teoria degli insiemi fuzzy. Dato che la generazione di tali modelli richiede che gli attributi continui siano discretizzati, è stata anche proposta una nuova soluzione distribuita per la discretizzazione di attributi continui basata sul concetto di entropia e, successivamente, tale soluzione è stata estesa con la logica fuzzy per la generazione di partizioni fuzzy. Infine, considerando la complessità dei modelli generati dalle precedenti soluzioni, è stato introdotto un approccio evolutivo distribuito per l'ottimizzazione delle prestazioni dei classificatori sia in termini di accuratezza che di interpretabilità.

Gli algoritmi proposti sono stati sviluppati secondo il paradigma di programmazione MapReduce ed eseguiti su noti framework per la processazione distribuita dei dati, ampiamente utilizzati sia in ambito di ricerca che industriale. La valutazione delle prestazioni è stata svolta attraverso l'utilizzo di differenti benchmark e i risultati ottenuti dagli approcci proposti e da altri algoritmi distribuiti di classificazione allo stato dell'arte sono stati ampiamente discussi in termini di accuratezza, complessità del modello e scalabilità.

Abstract

Classification problems have been widely studied in the context of data mining and different approaches to address these problems have been developed in the last decades. Among them, associative classification and decision trees have proved to be very effective and have been successfully employed in several application domains. Furthermore, some of these approaches have integrated the fuzzy set theory with the objective of dealing with uncertain and noise data. Unfortunately, most of the approaches proposed up to now have been designed for maximizing accuracy, often neglecting the complexity both in terms of memory that execution times. Thus, these approaches are generally not able to handle adequately the so-called “big data”.

In this Ph.D. thesis, we propose different solutions in a distributed environment for generating accurate and interpretable classification models for big data. In particular, we focus on associative classification and decision trees, integrating our solutions with fuzzy set theory. Since the generation of such models requires that continuous features are discretized, we also propose a novel distributed discretization approach based on information entropy. This approach has been therefore extended with fuzzy logic for generating fuzzy partitions. Finally, considering the complexity of the models generated by previous solutions, we propose a distributed evolutionary approach for optimizing both accuracy and interpretability of the classifiers.

The proposed algorithms are shaped according to the MapReduce programming model and have been deployed on well-known data processing frameworks, widely employed in research as well as industrial contexts. The performance evaluation has been carried out by using different big data benchmarks and the results obtained by the proposed approaches and by some state-of-the-art distributed classification algorithms have been extensively discussed in terms of accuracy, model complexity, and computation time.

To my beloved and lovely sister Stefania,

for being able to support not only herself but also other people
with her strength, courage and presence despite the dark times.

Contents

1 Introduction . . . 1

2 Big Data: technologies and state-of-the-art . . . 7
2.1 Distributed Processing Frameworks . . . 8
2.1.1 MapReduce . . . 9
2.1.2 Apache Hadoop . . . 10
2.1.3 Apache Spark . . . 14

3 Associative Classification . . . 17
3.1 Associative Classifiers . . . 18
3.1.1 Notation . . . 20
3.2 MRAC: a MapReduce Solution for Associative Classification of Big Data . . 22
3.2.1 Parallel FP-Growth . . . 22
3.2.2 The distributed approach . . . 23
Discretization . . . 23
CAR Mining . . . 27
Rule Pruning . . . 33
Classification . . . 35
3.2.3 MRAC+: a faster version of MRAC . . . 36
3.2.4 Experimental Study . . . 38
Performance of MRAC+ and MRAC . . . 39
Scalability analysis . . . 42
Tackling the dataset size . . . 46
3.3 Fuzzy Associative Classifiers . . . 48
3.3.1 Fuzzy Rule Based Classifiers . . . 50
3.3.2 Fuzzy association rules for classifications . . . 51
3.4 AC-FFP: a novel Associative Classification model based on a Fuzzy Frequent Pattern mining algorithm . . . 52
3.4.1 The Proposed Approach . . . 54
Discretization . . . 54

Fuzzy CAR Mining ... 56
Pruning ... 59
Classification ... 63
3.4.2 Experimental Study ... 64
3.5 DAC-FFP: a Distributed implementation of AC-FFP for Big Data ... 67
3.5.1 The Distributed Approach ... 68
Distributed Fuzzy Partitioning ... 69
Distributed Fuzzy CAR Mining ... 73
Distributed Fuzzy CAR Pruning ... 77
Reasoning methods ... 79
3.5.2 Experimental Study ... 80
Analysis of the fuzzy associative classifier performance ... 81
Scalability Analysis ... 84

4 Tree based Classification ... 89
4.1 Fuzzy Decision Trees ... 90
4.1.1 Background ... 93
4.2 The Proposed Algorithms ... 95
4.2.1 Fuzzy Partitioning ... 95
4.2.2 Fuzzy Decision Tree Learning ... 98
4.2.3 The Distributed Approach ... 103
4.3 Experimental Study ... 110
4.3.1 Performance analysis ... 110
4.3.2 Scalability analysis ... 115
4.3.3 Dealing the dataset size ... 117

5 Multi-Objective Evolutionary Fuzzy System for Big Data ... 121
5.1 Distributed MOEA: state-of-the-art ... 122
5.2 The Proposed Algorithm ... 123
5.2.1 PAES-RCS ... 123
5.2.2 The Distributed Approach ... 124
5.3 Experimental Study ... 127
5.3.1 Performance of DPAES-RCS: accuracy and complexity ... 128
5.3.2 Scalability analysis ... 132

6 Conclusions ... 135

References ... 139

List of Figures

2.1 The overall MapReduce Flow (R=2). 10
2.2 A MapReduce Job flow executed in top of Apache Hadoop. 12
2.3 A distributed Spark application flow executed on top of Apache Spark. 14

3.1 The Discretization step of the MapReduce Associative Classifier. 25
3.2 Pseudo-code of the first MapReduce phase of the discretization process. . . 26
3.3 Pseudo-code of the second MapReduce phase of the discretization process. 27
3.4 The CAR Mining step of the MapReduce Associative Classifier. 29
3.5 The MapReduce Parallel Counting Phase 30
3.6 The MapReduce Parallel FP-Growth Phase 31
3.7 A simple example of the Parallel FP-Growth execution. 32
3.8 The MapReduce Candidate Rule Filtering Phase 33
3.9 The Pruning step of the MapReduce Associative Classifier. 34
3.10 The MapReduce Training Set Coverage Pruning Job 35
3.11 The overall workflow of MRAC (on the left) and MRAC+ (on the right). 37
3.12 Runtime and Speedup for MRAC+ (a-b) and MRAC (c-d) on the overall Susy dataset. 44
3.13 Speedup of the discretization and the two main *learning* phases 46
3.14 Average Runtime of both MRAC+ (a) and MRAC (b) on the Susy dataset, varying the dataset size and the number of available cores. 47
3.15 An example of strong fuzzy partition obtained by the fuzzification of the output of the Fayyad and Irani's discretizer. 56
3.16 The fuzzy partitions of each variable in the example. 57
3.17 The FP-tree generated by using the example dataset. 59
3.18 Pseudo-code of the fuzzy CAR mining process based on FP-Growth. 60
3.19 Pseudo-code of the first type of pruning. 62
3.20 Pseudo-code of the third type of pruning. 63
3.21 The overall distributed Fuzzy Partitioning of the Fuzzy Decision Tree. 72
3.22 The overall FCAR Mining process of the DAC-FFP algorithm. 74

3.23 The overall FCAR Pruning process of the DAC-FFP algorithm. 78
3.24 Execution time and speedup of DAC-FFP on SUS dataset for increasing numbers of cores 86
3.25 Speedup of the different phases of DAC-FFP on SUS dataset for increasing number of cores 87

4.1 An example of fuzzy partition defined on the third bin boundary $b_{f,3}$. We suppose that the domain $[l_f, u_f]$ of S_f has been split into eight equi-freqnecy bins identified by seven bin boundaries $\{b_{f,1}, ..., b_{f,7}\}$ 97
4.2 An example of application of the recursive procedure to the fuzzy partition shown in Fig. 4.1 ($b^0_{f,l_{max}} = b_3$). 98
4.3 An example of multiple splitting on a continuous attribute with five triangular fuzzy sets. The blue circle shows an example of how a given example contributes to the cardinality computation. 99
4.4 An example of binary split performed by FBDT on a continuous attribute partitioned by five triangular fuzzy sets. 101
4.5 The overall distributed Fuzzy Partitioning of the FDT. 105
4.6 The overall DFDT Learning approach. 108
4.7 Speedup of Fuzzy Partitioning (a) and FBDT Learning (b) on the overall Susy dataset, varying the number of cores. 116
4.8 Runtime (in seconds) of Fuzzy Partitioning (a) and Learning (b) on the Susy dataset, varying the dataset size. 118

5.1 The distributed candidate rules generation phase. 125
5.2 The distributed evolutionary optimization phase. 126
5.3 Average Pareto fronts obtained from each dataset. 130
5.4 Speedup of DPAES-RCS varying the number of cores 133
5.5 Execution time of DPAES-RCS varying the number of cores 133

List of Tables

3.1 Big datasets used in the experiments. 38
3.2 Values of the parameters for each algorithm used in the experiments. 40
3.3 Average accuracy $\pm$ standard deviation achieved by MRAC+, MRAC, Decision Tree (DT), and Random Forest (RF). 41
3.4 The computation times (in seconds) for the learning process in MRAC+, MRAC, Decision Tree (DT), and Random Forest (RF). 41
3.5 Complexities of MRAC+, MRAC, Decision Tree (DT), and Random Forest (RF). 42
3.6 Runtime, speedup (σ_6), and utilization ($\sigma_6(Q)/Q$) for MRAC+ and MRAC on the Susy dataset. 43
3.7 Runtime, speedup (σ_6), and utilization ($\sigma_6(Q)/Q$) of the *Discretization*, *Parallel FP-Growth*, and *Training Set Coverage Pruning* phases in the Susy dataset. 45
3.8 Average runtime of MRAC+ and MRAC on the Susy dataset, varying the dataset size and the number of available cores. 47
3.9 A simple dataset characterized by four input features. 57
3.10 The fuzzy supports of each fuzzy set in the example. 58
3.11 The fuzzy values associated with the highest membership degree and the corresponding fuzzy objects for each pattern in the example dataset. 58
3.12 Datasets used in the experiments (sorted for increasing numbers of input variables). 64
3.13 Values of the parameters for each algorithm used in the experiments. 65
3.14 Average results obtained by CMAR, FARC-HD, D-MOFARC and AC-FFP. . . 66
3.15 Results of the Wilcoxon signed-rank test with a significance level $\alpha = 0.05$. . 67
3.16 Big datasets used in the experiments. 81
3.17 Values of the parameters for each algorithm used in the experiments. 82
3.18 Average accuracy $\pm$ standard deviation achieved by DAC-FFP, MRAC+ and MRAC. 82
3.19 Complexities of DAC-FFP, MRAC+ and MRAC. 84

3.20 The computation times (in seconds) for the learning process in DAC-FFP, MRAC+, MRAC. 84
3.21 Speedup of the overall algorithm for the Susy dataset. 85
3.22 Runtime and speedup of each DAC-FFP phase in the Susy dataset. 86

4.1 Big datasets used in the experiments. 110
4.2 Values of the parameters for each algorithm used in the experiments. 111
4.3 Average accuracy $\pm$ standard deviation achieved by FMDT, FBDT and DDT. 112
4.4 Complexities of FMDT, FBDT and DDT. 112
4.5 Complexities of Fuzzy Partitioning for both FMDT and FBDT. 113
4.6 The execution time (in seconds) for FMDT, FBDT and DDT. 114
4.7 Run-time, speedup (σ_8), and utilization ($\sigma_8(Q)/Q$) of both Fuzzy Partitioning and FBDT Learning processes for the Susy dataset. 116
4.8 Run-time (in seconds) of FBDT on the Susy dataset, varying the dataset size. 117

5.1 Datasets used in the experiments. 128
5.2 Values of the parameters used in the experiments for DPAES-RCS. 128
5.3 Values of the parameters used for the C4.5 and average number of rules and features in the rule bases extracted from the generated trees. 129
5.4 Average results obtained by the FIRST, MEDIAN and LAST solutions generated by DPAES-RCS 129
5.5 Results of the application of DPAES-RCS and of the Decision Tree implemented in MLlib. 131
5.6 Results of the Wilcoxon signed-rank test on the CRs obtained on the test sets by DPAES-RCS and DT 131
5.7 Speedup of the PAES-RCS algorithm in classification. 133

1 Introduction

Object classification is one of the most studied data mining paradigms [57, 79, 197] and consists of assigning a *class label* to an object described by a set of *features*. The classification task is carried out by using a specific model, namely the *classifier*, previously built by using a set of training examples. A large amount of different model types have been proposed in the literature, such as decision trees, rule-based models, probabilistic models, SVM, instance-based models, and neural networks. Furthermore, some works have exploited the use of fuzzy set theory to deal with uncertainty and build noise-tolerant classifiers. All these approaches have been often trained only on a few examples, at most thousands. The today classification applications, however, deal with a much larger number of examples and should produce results in a reasonable time. Indeed, nowadays, more than 1 exabyte is generated every day and organizations and researchers must be able to extract information in a few minutes by processing billions of data collected from different sources. Thus, scalability is one of the major issues that should be addressed [132, 148].

Dealing with *Big Data*, that is, datasets whose size is beyond the ability of typical database software tools to capture, store, manage and analyze [102, 104, 126, 138], most of the classification learning algorithms proposed in the literature are practically inapplicable [106, 148]. Indeed, these algorithms have been generally designed for maximizing the accuracy, often neglecting computational complexity and memory usage [21, 30]. New scalable and distributed data mining techniques as well as parallelization of traditional algorithms have to be designed and implemented to address big data [119, 150, 177, 200, 204].

The most fundamental challenge for Big Data applications is to explore the large volumes of data and extract useful information or knowledge [156, 199] so that companies can deploy advanced analytics solutions for providing an insight of their businesses, planning better decisions, and outperforming the competition [9, 111]. A simplistic solution to deal with massive data relies on selecting only a subset of data objects, so that traditional data mining algorithms could be executed in a reasonable time. However, downsampling techniques may delete useful knowledge [17]. Thus, approaches that consider the overall

dataset are far more desirable, and in our context this means explicitly addressing Big Data.

So far, data mining with massive data is intrinsically related to the open source software revolution of the last years [199]. Most of the works proposed in the literature deal with Apache Hadoop, a cluster computing framework for data-intensive distributed applications, based upon the MapReduce programming model and a distributed file system. Indeed, Hadoop allows running custom applications that rapidly process large datasets in parallel on a cluster of machines, taking care of communications among them and possible failures, and efficiently handling network bandwidth and disk usage. As regards classification problems, some recent works have proposed several distributed MapReduce solutions of classical algorithms, such as SVM [8, 27, 82], KNN [209], boosting [150], Random Forest [48, 105, 176], decision tree [43, 129], naive Bayes, neural network backpropagation, and logistic regression, investigating performance in terms of speedup [38]. However, only few authors integrate fuzzy logic theories in their parallel and distributed models for handling big datasets for classification problems. For instance, in [19] the authors propose a classification algorithm based on an optimized version of the Fuzzy c-Means clustering algorithm, which partitions the big data into various subsets. The algorithm retrieves clusters on the different subsets and a classifier based on Bayesian theory assigns labels to these clusters and predicts unlabeled instances. Moreover, in the last years, parallel versions of genetic/evolutionary fuzzy systems, that is, fuzzy models designed by means of genetic/evolutionary algorithms, have been recently proposed in the literature, both for generating fuzzy association rules [85] and Fuzzy Rule Based Classifiers (FRBCs) [90, 160].

Actually, the first contributions in the context of distributed FRBCs under the MapReduce paradigm have been proposed by the researchers of the University of Granada. In [119], the authors describe Chi-FRBCS-BigData, a fuzzy rule-based classification system based on the Chi et al.'s approach [36]. The algorithm has been modified to deal with big data using a two-stage MapReduce approach. The first stage builds the model from chunks of the training set: a group of fuzzy rules is generated from each chunk. Then, these groups are fused together in the reduce phase. The second stage estimates the class using the model learned in the first stage. The authors have investigated different approaches for fusing the fuzzy rules by developing two different versions, named Chi-FRBCS-BigData-Max and Chi-FRBCS-BigData-Ave. Moreover, an improved version, called Chi-FRBCS-BigDataCS, has been proposed in [119] for handling imbalanced big datasets. In [58], authors have proposed FDT 2.0, an improved version of FDT, obtained by integrating the FDT approach into the modern database technology for improving the scalability of the overall algorithm and handling large data sets. However, the tests have been carried out on datasets, which involve at most 400,000 instances, using MySQL as modern database manager. Thus, FDT 2.0 cannot be considered a solution for managing big datasets.

Generally, classification algorithms proposed in the literature require that a fuzzy partition is defined on each continuous attribute before starting the tree learning: the partition

is generally obtained by adopting heuristic approaches, which optimize purposely-defined indexes [190, 191, 203]. Actually, the discretization process is crucial to performance of FDTs, especially when a huge amount of data is involved. Recently, in [157, 158] a distributed implementation of the well-known Entropy Minimization Discretizer [61] has been proposed in order to partition continuous attributes.The algorithm first distributes the computation of the class frequency for each attribute, then sorts the values of each attribute in ascending order and selects a set of interval boundaries by retrieving those values that fall in the class borders. Finally, the generation of cut-points based on the entropy information is performed. For all attributes characterized by a low number of boundary values (lower than a fixed threshold), the computation can be processed independently in a single step. On the other hand, for attributes characterized by a high number of boundary values, the selection of the best cut-points has to be carried out iteratively. The first case is obviously more efficient. The second case, although less efficient, happens more rarely. The algorithm has been tested by using two datasets (up to about 65 millions of instances) and the generated cut points have been employed by the distributed Naive Bayes classifier available on the MLlib Library.

Although with the increasing interest on the big data phenomenon, researchers are continuously investigating new algorithms, taking into account not only the accuracy of the classifiers, but also the scalability of the proposed approaches, there is still a lack on several data mining applications, especially for classification problems. As described in the previous paragraph, so far the works have been focused only on a few of the classical algorithms. However, in the last years, different strategies such as classifier based on association rule mining techniques and decision trees have proved to be very effective [24, 79, 101, 108, 113, 114, 154, 155, 162, 197, 201] in terms of classification rate and interpretability, outperforming most of the classical algorithms proposed in the literature. Moreover, fuzzy set theory has been successfully employed for building more accurate and robust models than their crisp versions [6, 33, 34, 63, 93, 116, 123, 125, 147, 149, 188]. On the other hand, the use of fuzzy set concepts makes the generation of the classifier models more complex, holding back their utilization on the big data context. Indeed, in case of fuzzy partitions, an input value can support more than one fuzzy set with different membership degrees; thus the amount of information described by the fuzzy approaches is generally bigger than the one described by crisp approaches. With this aim, in this PhD. thesis, we propose different distributed solutions for building accurate and interpretable classification models that are able to deal with big datasets. In particular, we focus on associative classifiers and decision trees approaches, integrating our solutions with the concepts of fuzzy set theory. Moreover, taking care about complexity of the generated models, we propose a distributed evolutionary approach for optimizing both accuracy and interpretability of the classifiers. To the best of our knowledge, no solution which employs such techniques and is able to handle millions of data has been proposed yet. Moreover, we point out that our approaches allow handling big datasets even with modest hardware support. For all the proposed algorithms, we present and discuss the experimental results obtained perform-

ing several simulations over big data benchmarks and we compare the performance in terms of classification rate, model complexity and computational time of our approaches with the ones achieved by other state-of-the-art distributed learning algorithms.

As regards associative classification, first we propose a distributed association rule-based classification scheme, denoted MRAC, shaped according to the MapReduce programming model. Second, we introduce a novel associative classification model, namely AC-FFP, which integrates the concepts of fuzzy set theory with the frequent pattern mining algorithm. Finally, we extend the AC-FFP algorithm with the MapReduce approach for distributing the computation over a cluster of machines, denoted DAC-FFP. At high level, the scheme mines classification association rules (CARs), using a properly enhanced, distributed version of the well-known FP-Growth algorithm. Once CARs have been mined, the proposed scheme performs a distributed rule pruning and the set of survived CARs is used to classify unlabeled patterns. To design and implement such algorithms, several novel strategies have been proposed for efficiently generating accurate classifiers. First, we have proposed a MapReduce approach based on the classical Fayyad and Irani discreitzation classifier [61] for discretizing continuous features. We have also extended the approach for generating fuzzy partitions. Second, we have introduced a MapReduce approach of the FP-Growth algorithm for generating association rules for classification. We have also proposed some design choices for speeding up the overall process. Fourth, we have extended the FP-Growth algorithm for mining fuzzy CARs. In particular, we have proposed a novel approach that is able to handle a higher amount of information than the one described in previous works [37, 109]. Third, after mining the frequent patterns and generating CARs and fuzzy CARs, we have carefully designed a distributed MapReduce implementation of the pruning step. Fourth, we have introduced two novel techniques for classifying unlabeled patterns by performing a ranking of the generated rules.

As regards fuzzy decision trees, we have proposed a distributed fuzzy discretizer and a distributed FDT (DFDT) learning scheme upon the MapReduce programming model for managing big data. The discretizer generates a Ruspini fuzzy partition for each continuous attribute by using a purposely adapted distributed version of the well-known method proposed by Fayyad and Irani in [61]. The fuzzy partitions computed by the discretizer are used as input to the DFDT learning algorithm. We adopt and compare two different versions of the learning algorithm based on binary and multi-way splits, respectively. Both the versions employ the information gain computed in terms of fuzzy entropy for selecting the attribute to be adopted at each decision node.

So far, to increase the accuracy of the classifiers, all the distributed strategies proposed in the literature have been characterized by a high number of rules, making such models not interpretable. With the aim to maximizing the accuracy and minimizing the complexity, we have proposed a distributed multi-objective evolutionary algorithm for generating fuzzy rule based classifiers. The experimental study has shown that the distributed version can efficiently extract compact rule bases with accuracy comparable to the one achieved by state-of-the-art algorithms.

The rest of the thesis is organized as follows. In Chapter 2, we provide an overview of Big Data and technologies that have been developed in the last years: we focus on MapReduce as programming model, and Apache Hadoop and Apache Spark as data processing frameworks. We exploit these technologies in our classification problems. In Chapter 3 we introduce the associative classification and we describe our MapReduce solutions based on a distributed association rule mining technique. In Chapter 4 we present the fuzzy decision trees and we detail our approach based on MapReduce. In Chapter 5 first we provide some preliminaries on multi-objective evolutionary algorithms and then we introduce our distributed approach for building classification models characterized by different trade-offs between accuracy and complexity. For each proposed algorithm, we present and discuss the experimental results obtained by performing simulations with the proposed approaches and with the comparison methods. Finally, in Chapter 6, we draw some conclusion about the works described in this thesis.

2

Big Data: technologies and state-of-the-art

Recently, the term “Big Data” has been coined referring to those challenges and advantages derived from collecting and processing vast amounts of data [131, 164]. Every day more than 1 exabyte (10^9 gigabytes) of data are generated and in the last two yeara 90% of the total data generated in history has been produced [199]. The rate of data creation is accelerating and this astonishing growth has profoundly affected businesses. Nowadays, organizations must deal with petabyte-scale of different data collected from several sources: click streams, blog posts, tweets, social network interactions, transaction histories, sensors, photo and so on [132].

Compared to traditional data, big data are characterized by multiple “Vs”, namely, huge Volume, high Velocity, high Variety, low Veracity, and high Value [95] (actually, in the very first works only the first three “Vs” were considered). Traditional data management and analysis systems, mainly based on relational database management systems (RDBMSs), are inadequate in tackling the big data challenges [86]. Specifically, the following two aspects deserve attention:

- From the data structure perspective, RDBMSs have been designed to deal mainly with structured data, and little support is provided to semi-structured or unstructured data.
- From the scalability perspective, RDBMSs usually scale up with expensive hardware and cannot scale out with commodity hardware in parallel; this approach is unsuitable to cope with ever-growing data volumes.

To address these challenges and obtain effective and timely data management, various ad-hoc solutions for big data systems have been proposed [35]: *Cloud computing*, which can be deployed as the infrastructure layer for big data systems to meet requirements on cost-effectiveness, elasticity, and ability to scale up/out; *Distributed file systems* and *NoSQL databases*, suitable for persistent storage and the management of massive scheme-free datasets; *MapReduce* and *Pregel*, programming models that simplifies the parallelization of the computation flow across large-scale clusters of machines [47]; *Cluster computing frameworks* like Apache Hadoop [77, 82] and Apache Spark [168, 206], which integrate data storage, data processing, system management, and other modules

to form comprehensive, powerful system-level solutions. The central role played in this field by supporting systems and related analysis techniques has lead to a new definition for Big Data: “Big Data represents the Information assets characterized by such a High Volume, Velocity and Variety to require specific Technology and Analytical Methods for its transformation into Value” [45].

In the following, we present an overview of the programming models and frameworks that have been introduced in the last years for addressing the Big Data challenges. Indeed, we focus on MapReduce (Section 2.1.1) for the programming model, and on Apache Hadoop (Section 2.1.2) and Apache Spark (2.1.3), the two most widespread execution environments employed so far. We exploit this technologies in our classification problems.

2.1 Distributed Processing Frameworks

During last years, different programming models have been proposed for simplifying the parallelization of the computation flow across large-scale clusters of machine. So far, the best known model is MapReduce [46], mainly due to its simplicity [175]. However, for specific classes of machine learning algorithms, high-level parallel abstractions like MapReduce are insufficiently expressive while other low-level tools like MPI (Message Passsing Interface) [124] leave experts repeatedly solving the same design challenges, forcing them to address specific issues related to machine hardware and data representation [122]. For instance, there are many problems that can be intuitively modeled using graphs with sparse computational dependencies that require multiple iterations to converge. With this aim, Google proposed in 2010 Pregel [128], a computational model for large-scale graph processing.

A typical graph-parallel problem is expressed using graphs with vertices and edges, where each vertex and edge have associated data with them. Programs are expressed as a sequence of iterations and at each iteration, vertex and edge data are updated and a bunch of messages are exchanged between neighboring entities, modifying the graph topology. In particular, a vertex can receive messages sent in the previous iteration, send messages to other vertices and modify its own state. This update function is typically the same for every vertex, and is written by the user. There may or may not be a synchronization step at the end of every iteration. The synchronization determines the level of parallelism that is based on the three consistency models: full, edge or vertex [122]. In a distributed environment, the graph is cut and divided across multiple machines and the communications between vertices is performed by exploiting the message passing paradigm.

As regards cluster computing frameworks, in the last years, several open source projects have been developed to deal with big data, thanks also to the contribution of companies such as Facebook, Yahoo!, Twitter and so on. Some examples are: Spark [168] and Flink [64], fast and general engines for large-scale and data processing; Dryad [53, 87] and Ciel [143], universal execution engines for distributed data-

flow computing; Apache S4 [145], a platform for processing continuous data streams; Storm [170], a software for streaming data-intensive distributed applications similar to S4, Spark Streaming [169] and Flink Streaming [66]; Dremel [135] and Apache Drill [52], scalable, interactive low latency ad hoc query systems for analysis of read-only nested data; Giraph [71], GraphLab [75, 74, 121, 122] and Pegasus [97], iterative graph processing systems built for high scalability.

As regards data mining tools for big data, Mahout [127, 148] is the most popular machine learning library running on top of Hadoop. It implements a wide range of machine learning and data mining algorithms for clustering, recommendation systems, classification problems, dimension reduction and frequent pattern mining. The MLlib library [140] supports similar features on Spark. Among other machine learning tools, we highlight MOA (Massive online Analysis) [141, 20], FlinkML [65] Vowpal Wabbit [183], and H20 [76].

2.1.1 MapReduce

In 2004, Google proposed the MapReduce programming framework [46, 47]. The framework divides the work into a set of independent tasks and parallelizes the computation flow across large-scale clusters of machines, taking care of communications among them and possible failures, and efficiently handling network bandwidth and disk usage. MapReduce is a programming model based on functional programming, designed for processing large volumes of data with parallel and distributed algorithms on a computer cluster. It divides the computational flow into two main phases: *Map* and *Reduce*. By simply designing Map and Reduce functions, developers are able to develop parallel algorithms that can be executed across the cluster. The overall computation is organized around $\langle key, value \rangle$ pairs: it takes a set of *input* $\langle key, value \rangle$ pairs and produces a set of *output* $\langle key, value \rangle$ pairs. Figure 2.1 shows the overall MapReduce flow.

When the MapReduce execution environment executes a user program, it automatically partitions the dataset into a set of Z independent *splits* that can be processed in parallel by different machines. While the number of map tasks is determined by the number Z of the input splits, the number R of reducers is defined by the user. Thus, there are Z map tasks and R reduce tasks which have to be executed. Furthermore, the MapReduce framework starts up many copies of the user program on the machine cluster. One copy is denoted as *master*: the master schedules and handles tasks within the cluster. The others are called *workers*. The master assigns a map task or a reduce task to any idle worker. When a map task is assigned to a worker, the worker reads the contents of the corresponding input split, parses the $\langle key, value \rangle$ pairs of the input data and passes the computational flow to the user-defined Map function. The Map function takes a single $\langle key, value \rangle$ pair as input and produces a list of intermediate $\langle key, value \rangle$ pairs as output. This process can be represented as:

$$reduce(key1, value1) \rightarrow list(key2, value2) \tag{2.1}$$

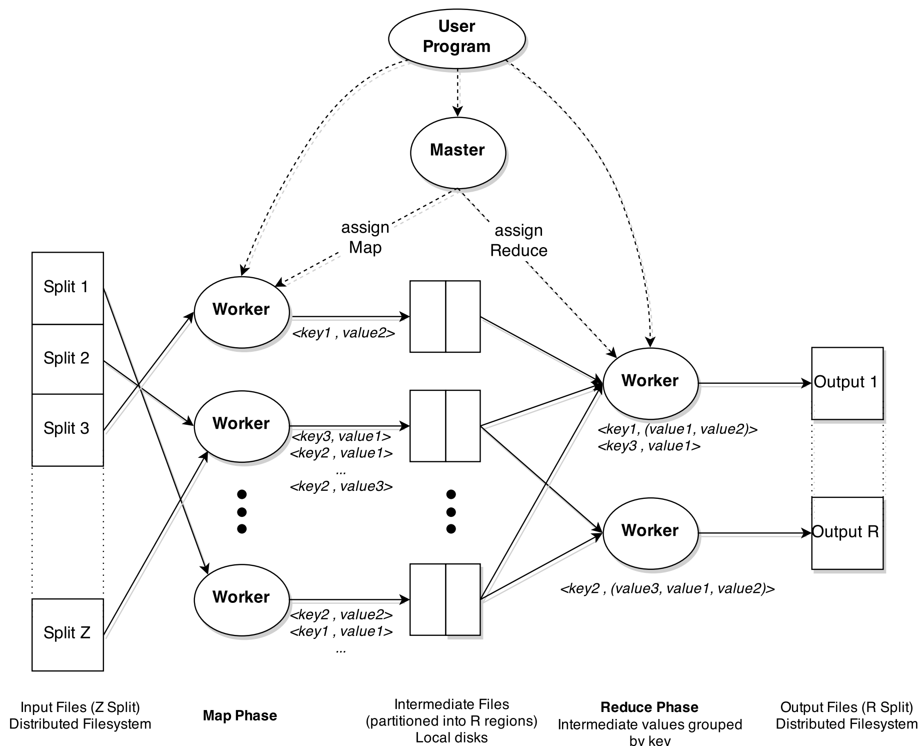


Figure 2.1: The overall MapReduce Flow (R=2).

Workers periodically store in the local disk the intermediate values produced by map functions and partition these values into R regions. Each region represents the intermediate subspace of the key space and is generated by a partitioning function (for instance $hash(key)$ **mod** R). Finally, these workers return back results to the master that is responsible for notifying reduce workers. When a reduce worker is notified, it reads remote intermediate data from the local disks of map workers, and groups and sorts them according to the intermediate key. Then, it iterates over the sorted intermediate data and, for each unique key, passes the computational flow to the Reduce function defined by the user. The Reduce function takes the key and the associated value list as input and generates a new list of values as output. This process can be summarized as:

$$map(key2, list(value2)) \rightarrow list(value2) \tag{2.2}$$

Finally, the output of the reduce function is appended to the output final file.

2.1.2 Apache Hadoop

So far, the most widespread execution environment for the MapReduce programming model has been Apache Hadoop [77, 194]. Hadoop allows the execution of custom applications that rapidly process big datasets stored in its distributed file system, called

Hadoop Distributed Filesystem (HDFS). It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures. The project includes three Java-based modules: (i) Hadoop Common, a set of utilities that support the other Hadoop modules; (ii) Hadoop Distributed File System (HDFS), a distributed file system that provides high-throughput access to application data; (iii) Hadoop MapReduce, a system for parallel processing of large data sets.

Inspired by Google File System [70], the HDFS is a distributed file system designed to run on commodity hardware and to reliably store very large files across machines in a large cluster around the idea of the most efficient streaming data processing pattern, namely "write-once, read-many-times". Indeed, HDFS stores each file as a sequence of blocks of the same size (by default 64MB). Each block is then replicated for fault tolerance and sent in different nodes among the cluster. This approach allows to store files that are larger than any single disk of each machine. HDFS is implemented following the master/worker architecture: a *Namenode* (the master) and several *Datanodes* (the workers). The Namenode is in charge to *(i)* manage the filesystem namespace, performing typical filesystem namespace operations such as opening, closing, and renaming of files, *(ii)* regulate the accesses to files by clients, and *(iii)* keep track of the status of the all Datanodes and the metadata of each file stored in the cluster. On the other hand, Datanodes manage the blocks attached to the nodes that they run on, performing some operations such as block creation, deletion, and replication upon instruction from the Namenode, and are responsible for serving read and write requests from filesystem clients.

Similar to HDFS, Hadoop MapReduce has a master/slave architecture. At high-level, it consists in a single master server or *JobTracker* and several slave servers or *TaskTrackers*, one per node in the cluster. The JobTracker is a Java-based application and represents the point of interaction between users and the framework. Users submit a so-called map/reduce *jobs* to the JobTracker, which puts them in a queue of pending jobs and executes them according to the FIFO (First-In First-Out) paradigm. The JobTracker assigns map and reduce tasks to the TaskTrackers. Each TaskTracker hosted in the slave servers is a Java-based application that executes the tasks which the job has been split into and handles data transfers between map and reduce phases.

Figure 2.2 illustrates how Hadoop runs a MapReduce job. The overall process can be summarized into five main sub-processes [194]: *(i)* job submission, *(ii)* job initialization, *(iii)* task assignment, *(iv)* task execution, and *(v)* job completion.

When the job submission process receives the user's MapReduce program (step 1), it asks to the JobTracker for a new job ID (step 2) so that each job can be identified within the cluster. Then, the process checks the input/output parameters of the job, computes the input splits, and copies all the resources needed to run the job into the JobTracker's filesystem (step 3) in a directory identified by the job ID. The resources include JAR file, job configuration file and the computed input splits, so that each TaskTracker knows how

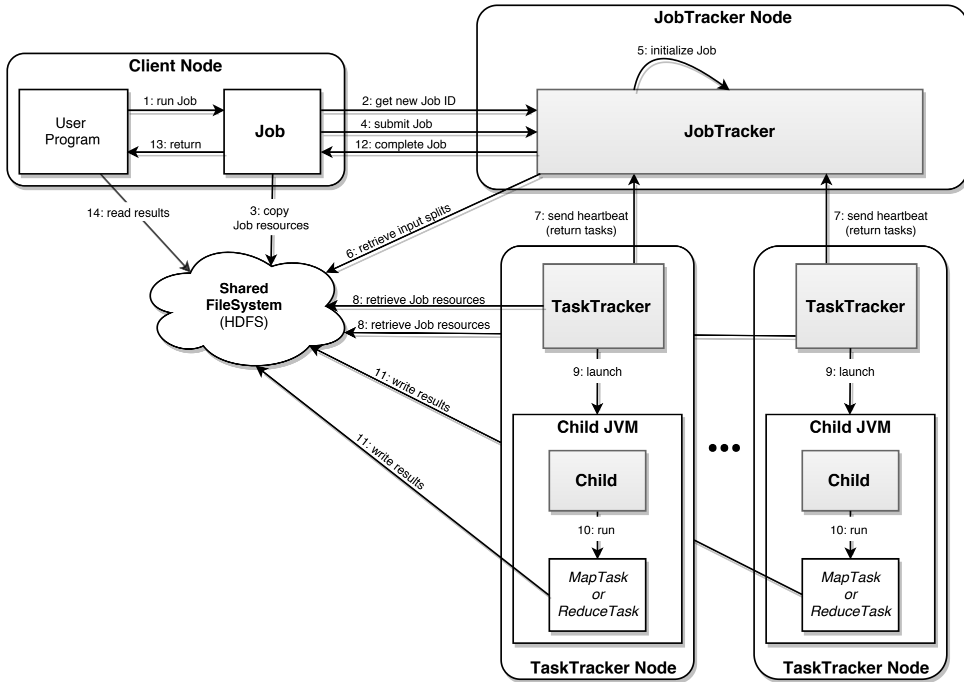


Figure 2.2: A MapReduce Job flow executed in top of Apache Hadoop.

to run each specific task. Finally, the last step of job submission process is devoted to notify the JobTracker that a new job is ready to be executed (step 4). At this point, the job is put into an internal queue from where the job scheduler will pick it up and initialize it. Initialization involves creating an object, which encapsulates its tasks and some information to keep track of the status and progress of each task (step 5). The list of tasks that must be executed (step 6) are retrieved by the JobTracker from the input splits computed in step 3. Then, one map task is created for each input split. Note that with the default parameters, each input split overlaps one HDFS block. A unique ID is given to each task, map or reduce. We highlight that at this point, two further tasks are created: a job setup task and a job cleanup task. These are executed by TaskTrackers and are used to run code to set up the job before any map task runs and to cleanup after all the reduce tasks are complete. Finally, the job scheduler chooses the job from the queue that must be run. There are several scheduling algorithms, but by default jobs are scheduled according to the FIFO paradigm. When a job has been selected, the JobTracker assigns the tasks for the job. Indeed, the TaskTrackers have a fixed number of slots for both map and reduce tasks. The default scheduler fills empty map task slots before each reduce task slots. For a map task, it picks a task whose input split is as close as possible to the TaskTracker, so that scheduler can take advantage of *data locality*, minimizing network congestion and increasing the overall throughput of the system. The assumption is that “moving com-

putation is cheaper than moving data" and this is especially true when the size of the data set is huge. On the other hand, for reduce tasks, the scheduler simply takes the next in the list, since no data locality considerations can be applied. TaskTrackers run a simple loop that periodically send a heartbeat to the JobTracker to notify that is alive and send some information about its status (step 7). Moreover, TaskTracker is in charge to run a task when it has been assigned to him. First, it copies the JAR file from the shared filesystem into its own filesystem (step 8), then it launches a new Java Virtual Machine (step 9) to run the task (step 10) so that any exception thrown by the user's functions do not affect the TaskTracker. Similar to the job, each task can perform setup and cleanup actions. In particular, each reduce task runs the user's reduce function and the cleanup action is used to commit the task. In case of file-based jobs, the results of the reduce task are written to the final output location into the shared filesystem (step 11). Each child communicates with its parent the progress of the task. The JobTracker combines all the data retrieved by the heartbeats of each TaskTracker and produces a global view of all the jobs. When the JobTracker receives a notification from the completion of the last task, it sets the job status to "successful" (step 12). Finally, the computation flow comes back to the user's program (step 13) and the results can be read from the HDFS (step 14) as specified in the configuration file of the job.

Apache Hadoop has proved to be very effective in terms of scalability and distributed processing. However, for very large clusters that involve more than 4,000 nodes, the architecture described in the previous paragraph begins to suffer from scalability bottlenecks. To address this issue, a group at Yahoo! begins to design a new generation of MapReduce, called YARN (or simply MapReduce 2). The idea is to split the responsibilities of the JobTracker into two new separate entities: a *resource manager* to handle the resources across the cluster and an *application master* to manage the lifecycle of jobs running on the cluster, taking care of task progress monitoring. The application master negotiates with the resource manager for the cluster resources and then runs the application inside to the so-called *containers*. Unlike to MapReduce 1, each program/application has its dedicated application master, making this architecture closer to original Google MapReduce implementation described in Section 2.1.1. The new architecture is more general than the previous one and allows working not only with MapReduce but also with other distributed computing models like Spark [206], Hama [78, 165], Giraph [71], and MPI [124].

Hadoop has been mainly optimized for one-pass batch processing of on-disk data, which makes it slow for interactive data exploration and more complex multi-pass analytics algorithms. Moreover, due to a poor inter-communication capability or inadequacy for in-memory computation [110, 206], Hadoop is not suitable for those applications that require iterative/online computations or memory intensive algorithms. Even if some approaches like Twister [178, 59, 60] and HaLoop [25] have tried to addressing these issues by implementing the concept of iterative MapReduce runtimes, in the last years Apache Spark is getting more and more popular because of its enhanced flexibility and efficiency.

2.1.3 Apache Spark

Apache Spark [168, 205, 206] is an open-source framework originally developed in the AMPLab at UC Berkley. It has emerged as the next generation big data processing tool due to both its enhanced flexibility and efficiency. Spark is a fast and general-purpose cluster computing system and provides an optimized engine for supporting general execution graphs, allowing employing different distributed programming models, such as MapReduce and Pregel. Unlike the disk-based MapReduce paradigm supported by Hadoop, Spark employs the concept of in-memory cluster computing, where datasets are cached in memory to reduce their access latency. Thanks to this feature, Spark has proved to perform faster than Hadoop [206], especially in case of iterative and online applications.

Apache Spark employs a master/slave architecture, with one central coordinator and many distributed *workers* [98]. At high level, a *Spark application* runs as a set of independent processes on the top of the dataset distributed over the cluster. Each application consists of one *driver program* (the coordinator) and several *executors* [98, 206] (the workers).

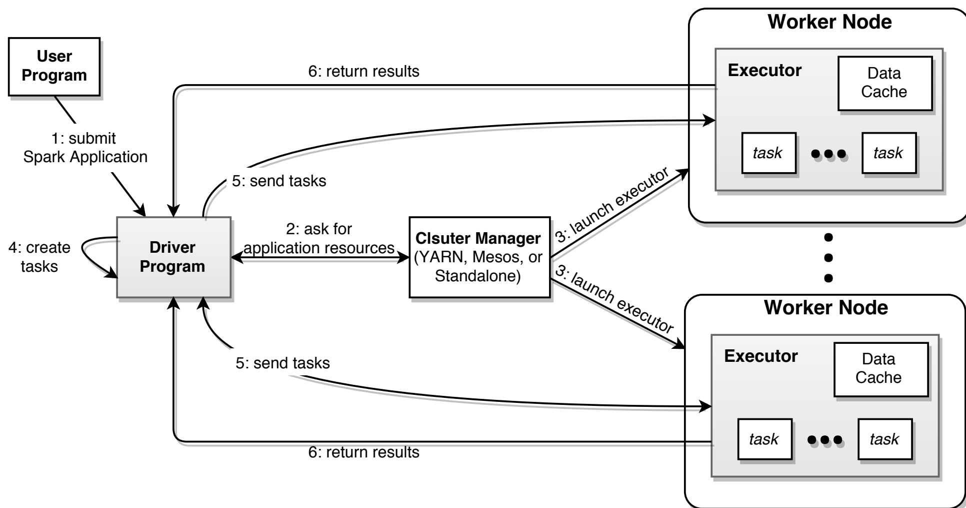


Figure 2.3: A distributed Spark application flow executed on top of Apache Spark.

As shown in Figure 2.3, when a Spark application is submitted (step 1), the driver program (also named Spark driver) acquires the resources on which its operations run by contacting an external service (step 2) named *cluster manager* that is in charge to launch the executors on behalf of the driver program (steps 3). By default, Spark adopts its on built-in service called Standalone, but it can work with two other popular open source cluster managers, namely Hadoop YARN (see Section 2.1.2) and Apache Mesos [137,

84]. The Spark driver, hosted in the master machine, is the process in charge to both run the user's main function and distribute *operations* on the cluster by sending several units of work, called *tasks*, to the executors. Indeed, to accomplish these duties, the Spark driver *(i)* converts the user program into several tasks by creating a logical *directed acyclic graph* (DAG) of operations, *(ii)* converts this logical graph into a physical execution plan, *(iii)* performs a different number of optimizations so that the physical execution plan can be converted into a set of *stages* (a set of multiple tasks, step 4), and *(iv)* schedules the task to each executor taking advantage of data locality (step 5). Each executor, hosted in a slave machine, runs several tasks in parallel, returns the results to the driver, and keeps data in memory or disk storage across them. Note that, in Spark ecosystem, a task represents the smallest unit of work. Spark architecture implies that applications are isolated from each other. Indeed, each application gets its own executor processes and tasks from different applications run in different JVMs. Thus, data cannot be shared across different Spark applications without writing it to an external storage system.

The main abstraction provided by Spark is the *Resilient Distributed Dataset* or simply (RDD) [205], which is a fault-tolerant collection of elements partitioned across the machines of the cluster that can be processed in parallel. These collections are resilient, because they can be rebuilt if a portion of the dataset is lost. RDDs support two types of operations: *transformations*, which create a new RDD from an existing one, and *actions*, which return a value to the driver program after running a computation on the RDD. The second abstraction provided by Spark is the *shared variable* that can be used in parallel operations. Sometimes, a variable needs to be shared across tasks, or between tasks and the driver program. To reduce the communication cost, Spark supports two types of shared variables for the two common usage patterns: *broadcast variables*, which can be used to cache a read-only variable in memory on each machine, and *accumulators*, which are variables for associative operations that allow implementing counters or sums.

3

Associative Classification

Associative classification integrates two of the most studied data mining paradigms, namely pattern classification and association rule mining [197]. Pattern classification deals with assigning a class label to an object described by a set of features. The classification task is carried out by using a specific model, namely the classifier, previously built by using a set of training examples. Association rule mining is the task of discovering correlation or other relationships among items in large database [2]. In the last years, association rule mining has become a very popular method to build highly accurate classification models. However, as stated in [149], such method suffers from some main weaknesses. First, the algorithms used for learning these classifiers are not able to adequately manage big data because complexity grows exponentially in terms of both time and memory with the number of training data objects. Second, association rule mining algorithms deal with binary or categorical itemsets, but real data objects are often described by numerical continuous features.

To overcome the first drawback, we propose a distributed association rule-based classification scheme, named MRAC, shaped according to the MapReduce programming model. The scheme mines classification association rules (CARs) using a properly enhanced, distributed version of the well-known FP-Growth algorithm. Once CARs have been mined, the proposed scheme performs a distributed rule pruning and the set of survived CARs is used to classify unlabeled patterns. As regards the second issue, we propose a novel associative classification model, namely AC-FFP, which integrates the concepts of fuzzy set theory with the frequent pattern mining algorithm. Moreover, its MapReduce approach, named DAC-FFP, is also described. We implemented our distributed solutions on the Hadoop framework and we tested our approaches with other state-of-the-art distributed learning algorithms on different real-world big datasets, comparing the results in terms of accuracy, model complexity, and computation time. Memory usage, computational time, scalability and achievable speedup for each phase of the learning process are also evaluated, by carrying out a number of experiments on small computer clusters. We highlight that the proposed approach allows handling big datasets even with modest hardware support. Moreover, we evaluated our fuzzy approach on

seventeen real-world small datasets and compared the achieved results with the ones obtained by using both a non-fuzzy associative classifier, namely CMAR, and two well-known fuzzy classifiers, namely FARC-HD and MOFARC-HD, based on fuzzy association rules. Using non-parametric statistical tests, we show that our approach outperforms CMAR and achieves accuracies similar to FARC-HD.

The chapter is organized as follows. Section 3.1 provides some preliminaries of associative classifiers. Section 3.2 describes the proposed MapReduce solution, with details of each single job, the experimental setup and the results achieved by the model. Section 3.3 provides an overview of fuzzy associative classifiers and introduces some notations for the fuzzy association rules. Section 3.4 and Section 3.5 describe the proposed fuzzy scheme and its distributed approach, respectively. Each section details the algorithms and comments the results in terms of accuracy, computation time, model complexity of the proposed solutions.

3.1 Associative Classifiers

Classifiers based on association relationships, generally known as *Associative Classifiers* (ACs), have proven to be very effective in classification problems. Several studies have shown that ACs have specific advantages over other traditional classification approaches such as Decision Tree and Rule Induction [192]. ACs are often capable of building efficient and accurate classification systems, since in the training phase they leverage association rule discovery methods that find all possible relationships among the attribute values in the training data set. This in turn leads to extract all hidden rules possibly missed by other classification algorithms. Notably, unlike Decision Trees, a rule in AC can be either updated or tuned without affecting the complete rule set, whereas in the Decision Tree approach the same task requires reshaping the whole tree [173]. Moreover, in different works [1, 16, 108, 113, 180, 201] it has been highlighted that ACs can achieve high classification performances and be more accurate than traditional algorithms such as C4.5 [155]. Also another advantage of using a classification based on association rules over ordinary classification approaches is that the output of an AC algorithm is represented by simple `if--then` rules, thus allowing the end-user to easily understand and interpret it [146, 171]. In last years, ACs have been successfully exploited in a number of real world applications such as phishing detection in websites [4], XML document classification [41], text analysis [202], and medical disease classification [54, 202].

Generally, an AC operates in three phases. In the first phase, a set of classification association rules (CARs) is mined from the training set. Preliminarily, the attribute values (items in the association rule context) characterized by an occurrence value beyond a given threshold, called *frequent items*, are selected from the dataset. Then, CARs are mined from frequent items. In the second phase, CARs are pruned according to support and confidence thresholds, and redundancy. Finally, the selected CARs are used to predict the class labels of input unlabeled patterns. The identification of fast

and efficient algorithms for association rule mining still represents a challenge for researchers [31, 32, 161, 182].

In the literature, a number of associative classification approaches, such as CBA [113], LB [136] and PCAR [31, 32] extract itemsets and then mine CARS exploiting the well-known Apriori algorithm [3]. This algorithm uses a “bottom up” approach, where candidate itemsets are generated by extending frequent itemsets one item at a time (a step known as candidate generation), and are tested against the overall dataset for evaluating if they are frequent. The algorithm terminates when no further successful frequent extension is possible. In many cases the Apriori candidate generate-and-test algorithm significantly reduces the size of candidate sets, leading to a good performance gain. However, the Apriori algorithm can suffer from two non-negligible costs [79]. Indeed, it may still need to generate a huge number of candidate itemsets, which have to be analyzed for verifying whether they are frequent. This requires to repeatedly scan the overall dataset. Since datasets are often very large, scanning the dataset is very expensive, in particular when the dataset cannot be stored in the main memory.

An interesting approach, which mines the complete set of frequent itemsets without generating all the possible candidate itemsets has been proposed in Han et al. [80]. The approach is called frequent pattern growth, or simply FP-Growth, and adopts a divide-and-conquer strategy. First, it compresses the dataset representing frequent items into a frequent pattern tree, or FP-tree, which retains the itemset association information. It then divides the compressed dataset into a set of conditional datasets (a special kind of projected datasets), each associated with a frequent item or a pattern fragment. For each pattern fragment, only its associated conditional dataset needs to be examined. Independently of the number of frequent items, FP-Growth scans the overall dataset only twice. On the contrary, Apriori can need several scans of the overall dataset. In [108] the authors have proposed the CMAR algorithm, an associative classification method based on the FP-Growth algorithm. CMAR exploits a tree structure to efficiently store and retrieve mined association rules. The authors demonstrate that CMAR outperforms the CBA algorithm both in terms of memory occupancy and execution time.

In addition to the methods for mining CARs based on Apriori and FP-Growth, in the literature we can find also the so-called GARC (Gain-based Association Rule Classification) approaches [29]. These approaches extend the Apriori-based algorithms as follows. First, they apply the information gain measure to select the best-split attribute for 1-itemsets: this attribute has to be included in the generation of all candidate k-itemsets. Second, they integrate the process of frequent itemset generation with the process of rule generation. Third, they define rule redundancy and rule conflicts, and incorporate corresponding strategies for rule pruning into the mining process. In the literature, GARC models have proved to outperform the CBA algorithm.

The different approaches proposed in the literature to generate ACs are focused on improvements of classification accuracy, often neglecting time and space requirements [146] for CARs generation. Dealing with *Big Data*, that is, datasets whose size is beyond the ability of typical database software tools to capture, store, manage and an-

alyze [104, 138], most of the classification learning algorithms proposed in the literature are practically inapplicable [106].

In the last years, different scalable and parallel implementations have been proposed for association rule mining. Most of these implementations have focused on the well-known FP-Growth algorithm. Some works have proposed to parallelize the FP-Growth algorithm by exploiting multiple threads on a shared memory environment [115, 207]. In [152], a distributed version that takes both data and computation across multiple machines into account, has been proposed. As pointed out in [106], in the approaches proposed in [152, 207] the high communication costs hamper an effective parallelization. Further, the multiple threads of the parallelized FP-Growth algorithm share the memory space. Thus, the problem of processing huge databases is not actually addressed by such solutions, since they do not eliminate the bottleneck of huge memory requirements. Other works investigate different parallel implementations, addressing communication cost, data placement strategies, memory and I/O utilization [26]. All these approaches are based on the MPI (Message Passing Interface) programming model [124] and achieve good performance in terms of scalability and speedup. However, as highlighted in [106], to push scalability to thousands or even more computers, we must significantly reduce the communication overheads between computers, and support automatic fault recovery as well. In particular, fault recovery becomes a critical problem in a massive computing environment, because the probability of no crash in thousands of computers employed in a single task execution is close to zero. The demands of sustainable speedup and fault tolerance require highly constrained and efficient communication protocols.

Currently, some popular cluster computing frameworks like Apache Hadoop [77, 82] and Apache Spark [168, 206] provide all these features. Although some recent works have proposed several parallel MapReduce solutions of classical frequent pattern mining algorithms such as FP-Growth [106] and Apriori [107, 112], no works have discussed the implementation of associative classifiers according to the MapReduce model.

3.1.1 Notation

Pattern classification consists of assigning a class C_l from a predefined set $C = \{C_1, \dots, C_L\}$ of classes to an unlabeled pattern. We consider a pattern as an F-dimensional vector of features. Let **X** $= \{X_1, \dots, X_F\}$ be the set of the F features and U_f, $f = 1, \dots, F$, be the universe of the fth feature. Features can be both continuous and categorical. Continuous features are discretized, that is, their universes are partitioned into contiguous intervals before performing the association rule mining process. Let $P_s = \{[a_{s,1}, a_{s,2}], (a_{s,2}, a_{s,3}], \dots, (a_{s,T_s}, a_{s,T_{s+1}}]\}$ be a partition of T_s contiguous intervals on the continuous feature X_s. Since each interval can be associated with a label, continuous features can be managed as categorical features. Let $V_f = \{v_{f,1}, \dots, v_{f,T_f}\}$ be the set of values associated with feature X_f, $f = 1, \dots, F$. In case of continuous features, each label $v_{f,j}$, with $j \in [1..T_f]$, identifies the jth interval $(a_{f,j}, a_{f,j+1}]$ of the partition of X_f.

Association rules are rules in the form $Z \to Y$, where Z and Y are sets of items. These rules describe relations among items in a dataset [79]. Association rules have been widely employed in the market basket analysis. Here, items identify products and rules describe dependencies among different products bought by customers [2]. Such relations can be used for decisions about marketing activities such as promotional pricing or product placements.

In the associative classification context, the single item is defined as the couple $I_{f,j} = (X_f, v_{f,j})$, where $v_{f,j}$ is one of the values that the variable X_f, $f = 1, ..., F$, can assume. A generic classification association rule CAR_m is expressed as:

$$CAR_m : Ant_m \to C_{l_m} \tag{3.1}$$

where Ant_m is a conjunction of items, and C_{l_m} is the class label selected for the rule within the set $C = \{C_1, \dots, C_L\}$ of possible classes. For each variable X_f, just one item is typically considered in Ant_m. Antecedent Ant_m can be represented more friendly as

$$Ant_m : X_1 \textbf{ is } v_{1,j_{m,1}} \ \dots \ \textbf{AND} \ \dots \ X_F \textbf{ is } v_{F,j_{m,F}} \tag{3.2}$$

where $v_{f,j_{m,f}}$ is the value used for variable X_f in rule CAR_m.

Let $T = (\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_N, y_N)$ be the training set, where, for each object $(\mathbf{x}_i, y_i)$, $x_{i,f}$, $i = 1, \dots, N$, is one of the discrete values that feature X_f, $f = 1, \dots, F$, can assume and $y_i \in C$. In case of continuous features, we replace the real value with the corresponding categorical value, that is, the value associated with the interval which the real value belongs to. We state that $\mathbf{x}_i$ matches a rule CAR_m if and only if, for each item $I_{f,j}$ in Ant_m with $f = 1, \dots, F$, the value $x_{i,f}$ of $\mathbf{x}_i$ has the same value $v_{f,j_{m,f}}$ of item $I_{f,j}$ for feature X_f in the rule.

In the association rule analysis, *support* and *confidence* are the most common measures to determine the strength of an association rule. Support of a classification association rule CAR_m, in short $supp(Ant_m \to C_{l_m})$, is the number of objects in the training set T matching antecedent Ant_m and having C_{l_m} as class label. Usually, the support value is normalized with the total number of objects. Support can be interpreted as the coverage of rule CAR_m in T and estimates the number of instances correctly classified by rule CAR_m.

Confidence of CAR_m, in short $conf(Ant_m \to C_{l_m})$, is the ratio between $supp(Ant_m \to C_{l_m})$ and the number of objects in T matching antecedent Ant_m. The value can be interpreted as the probability of correctly classifying the class label C_{l_m} in the unlabeled pattern $\widehat{\mathbf{x}}$ under the condition that $\widehat{\mathbf{x}}$ matches Ant_m. Formally, support and confidence can be expressed for a classification association rule CAR_m as follows:

$$supp(Ant_m \to C_{l_m}) = \frac{supp(Ant_m \cup C_{l_m})}{N} \tag{3.3}$$

$$conf(Ant_m \to C_{l_m}) = \frac{supp(Ant_m \cup C_{l_m})}{supp(Ant_m)} \tag{3.4}$$

where N is the number of objects in T, $supp(Ant_m \cup C_{l_m})$ is the number of objects in T matching pattern Ant_m and having C_{l_m} as class label and $supp(Ant_m)$ is the number of objects in T matching pattern Ant_m.

An AC is also characterized by its *reasoning method*, which uses the information from the rule base to determine the class label for a specific input pattern. In our AC scheme, we have experimented both the $weighted\ \chi^2$ and the best rule reasoning methods.

3.2 MRAC: a MapReduce Solution for Associative Classification of Big Data

In Section 3.1 we have pointed out that the current implementations of associative classifiers are not able to manage big data. In this section, we describe a distributed association rule-based classification scheme shaped according to the MapReduce programming model. The scheme mines classification association rules (CARs) using an enhanced, distributed version of the well-known FP-Growth algorithm, purposely adapted to solve classification problems. Once CARs have been mined, the proposed scheme performs a distributed rule pruning. The set of survived CARs is used to classify unlabeled patterns. Referring to an implementation on Hadoop, we show memory usage and time complexity for each phase of the learning process. We discuss two distinct versions of the classifier, which differ for the pruning and the inference mechanism.

We adopt seven real-world big datasets with different numbers of instances (up to 11 millions) to analyze scalability and speedup of each parallel job according to different work units and problem sizes. Further, focusing on accuracy, model complexity and computation time, we compare the results obtained by our approach with those achieved by two state-of-the-art distributed learning algorithms, namely the Random Forest implemented in Mahout on Hadoop and the Decision Tree implemented in Spark.

The rest of the section is organized as follows. Section 3.2.1 describes the distributed Mahout FP-Growth implementation (over Hadoop) that we properly modified and extended to serve as a component of our distributed associative classifier. We exploit this implementation in our distributed associative classifier. Section 3.2.2 and Section 3.2.3 describe the two proposed approaches, with details of each single job that runs on the cluster of machines. Section 3.2.4 presents the experimental setup and discusses the results in terms of accuracy, computation time, model complexity, speedup, and scalability.

3.2.1 Parallel FP-Growth

To mine the CARs from the training set, we adopt the well known FP-Growth mining algorithm. In the Hadoop-based version of our proposed classification scheme, we purposely modified the FP-Growth implementation available on Mahout. Such an implementation is based on the Parallel FP-Growth (PFP) proposed by Li et al. [106] for an efficient parallelization of the frequent patterns mining without generating candidate itemsets. The PFP algorithm breaks down a large-scale mining task into independent, parallel tasks

and uses three MapReduce phases to generate frequent patterns. In the first phase, the algorithm counts the support values for all the dataset items. In the second phase, each node builds a local and independent tree and recursively mines the frequent patterns from it. Such a subdivision requires the whole dataset to be projected onto different item-conditional datasets. An item-conditional dataset $T(I_{f,j})$, also called *item-projected dataset*, is a dataset restricted to the objects where the specific item $I_{f,j}$ occurs. In each object of the $T(I_{f,j})$, named *item-projected object*, the items with support smaller than $I_{f,j}$ are removed, and the others are sorted according to the descending support order. Since the FP-tree building processes are independent of each other, all the item-projected datasets can be distributed over the nodes and processed independently.

In the last phase, the algorithm aggregates the previously generated results and, for each item, selects only the highest supported patterns. As shown by empirical studies, the PFP algorithm achieves a near-linear speedup [106].

3.2.2 The distributed approach

The MapReduce Associative Classifier (MRAC) can be viewed as an extension of the well-known CMAR [108] algorithm in a distributed execution environment and consists of the following three steps:

1. *Discretization*: a partition is defined on each continuous feature by using a MapReduce discretization approach based on the algorithm proposed by Fayyad and Irani in [61];
2. *CAR Mining*: a purposely adapted version of the PFP algorithm is exploited to extract frequent CARs with support, confidence, and chi-squared higher than pre-fixed thresholds;
3. *Pruning*: rule pruning based on redundancy and training set coverage is applied to generate the final rule base.

In the following, we discuss the three steps in detail.

Discretization

The discretization of continuous features is a critical aspect in AC generation, and so far several different heuristic algorithms have been proposed to this aim [39, 51, 61, 100]. For MRAC we use the method proposed by Fayyad and Irani in [61]. This supervised method has been already adopted in CMAR and has proven to be very effective [108]. It determines the cut-points by exploiting the class information entropy of candidate partitions.

Let $T_{f,0} = [x_{1,f}, ..., x_{N,f}]^T$ be the projection of the training set T along variable X_f and let $a_{f,r}$ be a cut point for the same variable. Let $T_{f,1}$ and $T_{f,2}$ be the subsets of points of the set $T_{f,0}$, which lie in the two intervals identified by $a_{f,r}$. The class information entropy of the discretization induced by $a_{f,r}$, denoted as $E(X_f, a_{f,r}; T_{f,0})$ is given by

$$E(X_f, a_{f,r}; T_{f,0}) = \frac{|T_{f,1}|}{|T_{f,0}|} \cdot Ent(T_{f,1}) + \frac{|T_{f,2}|}{|T_{f,0}|} \cdot Ent(T_{f,2}) \quad (3.5)$$

where $|\cdot|$ denotes the cardinality and $Ent()$ is the entropy calculated for a set of points [61]. The cut point $a_{f,min}$ that minimizes the class information entropy over all possible binary partitions of $T_{f,0}$, is selected. The method is then recursively applied to both the intervals induced by $a_{f,min}$ until the following stopping criterion based on the Minimal Description Length Principle is achieved:

$$Gain(X_f, a_{f,min}; T_{f,0}) < \frac{\log_2(|T_{f,0}| - 1)}{|T_{f,0}|} + \frac{\Delta(X_f, a_{f,min}; T_{f,0})}{|T_{f,0}|} \quad (3.6)$$

where

$$Gain(X_f, a_{f,min}; T_{f,0}) = Ent(T_{f,0}) - E(X_f, a_{f,min}; T_{f,0}), \quad (3.7)$$

$$\Delta(X_f, a_{f,min}; T_{f,0}) = \log_2(3^{k_0} - 2) - [k_0 \cdot Ent(T_{f,0}) - k_1 \cdot Ent(T_{f,1}) - k_2 \cdot Ent(T_{f,2})] \quad (3.8)$$

and k_i is the number of class labels represented in the set $T_{f,i}$.

The method outputs, for each feature, a set of cut points. Let $U_f = [x_{f,l}, x_{f,u}]$ be the universe of variable X_f. Let $A_f = \{a_{f,1}, \ldots, a_{f,T_{f+1}}\}$, with $\forall r \in [1, \ldots, T_f], a_{f,r} < a_{f,r+1}$, be the set of cut points, where $a_{f,1} = x_{f,l}$ and $a_{f,T_{f+1}} = x_{f,u}$. Then, the method identifies the set $\{[a_{f,1}, a_{f,2}], \ldots, (a_{f,T_f}, a_{f,T_{f+1}}]\}$ of contiguous intervals, which partition the universe of variable X_f. If no cut point has been found by the algorithm for feature X_f, then no interval is generated for such a feature, and X_f is then discarded. We associate each interval $(a_{f,r}, a_{f,r+1}]$, $r \in [1, \ldots, T_f]$ with a categorical value $v_{f,r}$. Each categorical value represents an item.

In order to manage a large amount of data, we propose an approximation of the Fayyad and Irani method. We implement the discretization process by using two MapReduce phases, as shown in Fig. 3.1. Here, CU stands for Computing Unit. In the first phase, each Mapper splits the data block into a number of equi-frequency bins: the number is determined as a percentage γ of the HDFS block size. In the experiments, we used $\gamma = 0.1\%$. The output of each Mapper is the sorted list of bin boundaries. The Reducers join all the sorted lists and, for each variable, generate a sorted list of bin boundaries. In the second phase, for each bin determined by a pair of consecutive bin boundaries in the list, the Mappers compute the percentage of instances belonging to the different classes. These percentages are used by the Reducers to compute the Fayyad and Irani discretization algorithm on the bins. To limit the number of possible items, the imposed termination condition corresponds to getting to a partition in which one of two intervals contains less than a percentage ϕ of the instances (in the experiments, we adopted $\phi = 2\%$). In practice, we verified that the creation of items with a smaller support leads to a classifier excessively specialized on the training set, penalizing the performance on the test set. Hence, lower values for ϕ increase the overall runtime with no real advantage.

The proposed distributed approach makes it feasible dealing with a very large number of instances: the equi-frequency bins used in the first phase permit us to reduce the data amount the Fayyad and Irani discretization is applied to. Obviously, the higher the

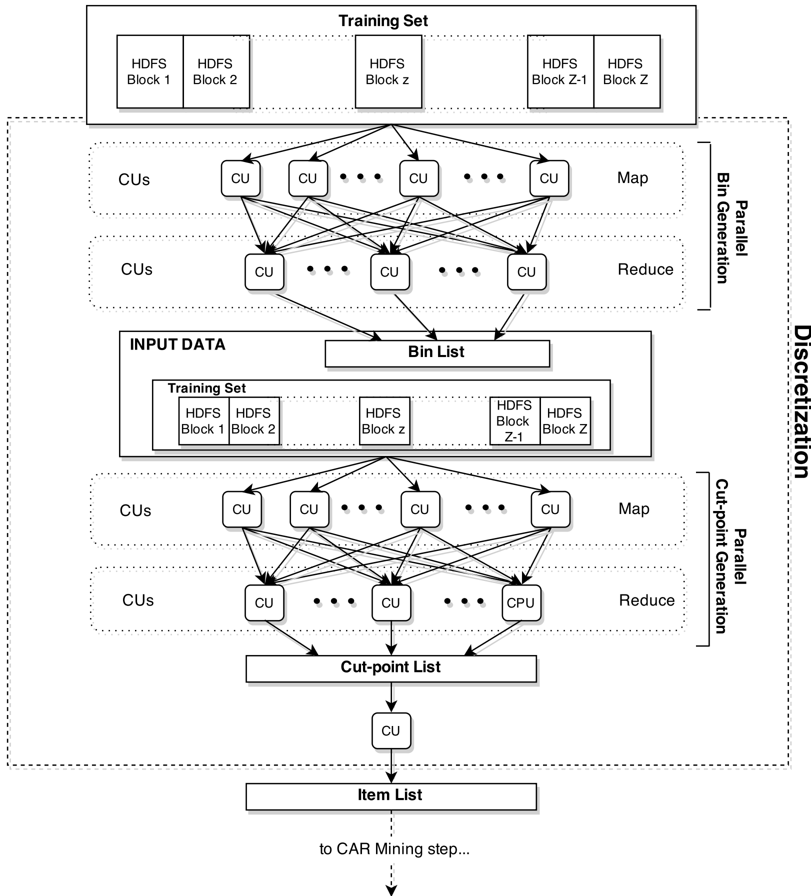


Figure 3.1: The Discretization step of the MapReduce Associative Classifier.

frequency used in the equi-frequency bins, the coarser the approximation in determining the cut-points for the Fayyad and Irani algorithm. However we must notice that, as we are managing millions of data, a difference of a few instances determined by choosing the cut-points between bin boundaries instead of the original instances is generally negligible in terms of achieved accuracy.

More specifically, the first MapReduce phase scans the dataset and computes at most $\Omega = Z \cdot \Gamma$ bin boundaries, where Z is the number of mappers and $\Gamma = 100/\gamma + 1$ is the number of bin boundaries. Let $B_{z,f} = \{b_{z,f_1}, \ldots, b_{z,f_\Gamma}\}$ be the sorted list of bin boundaries for the fth feature extracted from the zth HDFS block. First, each mapper loads all objects of the HDFS block into the main memory: the input key-value pair is represented by $\langle key, value = o_i \rangle$, where o_i is an object of the training set block. Then, for each continuous feature, the mapper computes the boundaries of equi-frequency bins, where each bin contains a number of instances equal to the percentage γ of the HDFS block size, and outputs a key-value pair $\langle key = f, value = B_{z,f} \rangle$, where f is the index

of the fth feature and $B_{z,f}$ is the list containing all the bin boundaries for the feature X_f extracted from the zth HDFS block. Each reducer is fed Z lists, denoted as $List(B_{z,f})$, of bin boundaries for each feature f and outputs $\langle key = f, value = B_f \rangle$, where $B_f = \{b_{f_1}, \ldots, b_{f_\Omega}\}$, with $\forall \omega \in [1, \ldots, \Omega - 1]$ $b_{f_\omega} < b_{f_{\omega+1}}$, is the sorted list of bin boundaries for feature f. The pseudo-code of the first MapReduce phase is shown in Fig. 3.2. Space and time complexities of the Map phase are $O(N/Q)$ and $O(F \cdot (N \cdot log(N/Q)/Q))$, respectively, where Q is the number of CUs. On the contrary, space and time complexities of the Reduce phase are $O(F \cdot \Omega/Q)$ and $O(F \cdot (\Omega \cdot log(\Omega))/Q)$ respectively.

```
 1: procedure MAPPER(key, value = o_i)
 2:     for all object o_i in HDFS − block do
 3:         dataBlock ← load(o_i);
 4:     end for
 5:     for all feature X_f in X do
 6:         if (ISCONTINUOS(X_f)) then
 7:             B_{z,f} ← COMPUTEEQUIFREQUENCYBINBOUNDARIES(dataBlock, f, γ);
 8:             Call OUTPUT(⟨key = f, value = B_{z,f}⟩);
 9:         end if
10:     end for
11: end procedure
12: procedure REDUCER(key = f, value = List(B_{z,f}))
13:     B_f ← NEWSORTEDLIST()
14:     for all B_{z,f} in List(B_{z,f}) do
15:         B_f ← INSERT(B_{z,f});
16:     end for
17:     Call OUTPUT(⟨key = f, value = B_f⟩);
18: end procedure
```

Figure 3.2: Pseudo-code of the first MapReduce phase of the discretization process.

The second MapReduce phase scans the dataset to compute the frequency of each class for each bin, and then performs the Fayyad and Irani algorithm on the bins. For each list B_f of sorted bin boundaries, the mapper first generates the set of bins $\{[b_{f_1}, b_{f_2}], \ldots, (B_{f_{\Omega-1}}, B_{f_\Omega}]\}$ from the bin boundaries in B_f. Then, it creates a vector $W_{z,f}$ of $\Omega - 1$ elements, where each element $W_{z,f,r}$ corresponds to the bin $(b_{f_r}, b_{f_{r+1}}]$. $W_{z,f,r}$ contains a vector of L elements, which stores, for each of the L classes, the number of instances of the class belonging to the r-th bin in the HDFS block. For each object o_i in the HDFS block, the mapper updates $W_{z,f}$. The input and output key-value pairs are $\langle key, value = o_i \rangle$ and $\langle key = f, value = W_{z,f} \rangle$, respectively. The reducer is fed a list $List(W_{z,f})$ of Z vectors $W_{z,f}$. The input key-value pair is $\langle key = f, value = List(W_{z,f}) \rangle$. The reducer creates a vector W_f of $\Omega - 1$ elements, where each element $W_{f,r}$ corresponds to the bin $(b_{f_r}, b_{f_{r+1}}]$ and contains the sum of the corresponding Z elements $W_{z,f,r}$. Thus, each element $W_{z,f}$ stores the frequency for each class in every bin along the whole dataset. Then, the reducer applies the Fayyad

and Irani discretization to the sequence of bins, setting the minimum frequency of each interval to ϕ. Finally, the reducer outputs $\langle key = f, value = A_f \rangle$, where A_f is the list of cut-points for the feature X_f. The pseudo-code of the second MapReduce phase is shown in Fig. 3.3. We would like to point out that the cut-points correspond to interval boundaries. Space and time complexities of the Map phase are $O(N/Q)$ and $O(N \cdot log(\Omega)/Q))$, respectively, where Q is the number of computing units (CUs). On the contrary, for the Reduce phase, space is $O(F \cdot (\Omega - 1)/Q)$ and time complexity depends on the execution time of the Fayyad and Irani algorithm, $O(F \cdot (FayyadAndIraniDis(\Omega - 1))/Q)$.

```
1: procedure MAPPER(key, value = o_i)
2:     B, W ← NEWLIST();                          ▷ List of B_f and W_{z,f} respectively
3:     for all index of feature f in F do
4:         B ← ADD(B_f);
5:         W_{z,f} ← INITIALIZE(size(B_f), L);
6:         W ← ADD(W_{z,f});
7:     end for
8:     o_i[] ← SPLIT(o_i);                         ▷ Array of F+1 values
9:     C_{l_i} ← GETCLASSLABEL(o_i[]);
10:    for all index of feature f in F do
11:        if (ISCONTINUOS(X_f)) then
12:            W ← UPDATEFREQUENCY(f, o_i[f], C_{l_i});
13:        end if
14:    end for
15:    for all feature X_f in X do
16:        if (ISCONTINUOS(X_f)) then
17:            Call OUTPUT(⟨key = f, value = W_{z,f}⟩);
18:        end if
19:    end for
20: end procedure
21: procedure REDUCER(key = f, value = List(W_{z,f}))
22:    W_f ← INITIALIZE();
23:    for all W_{z,f} in List(W_{z,f}) do
24:        W_f ← UPDATEFREQUENCY(W_{z,f});
25:    end for
26:    A_f ← FAYYADANDIRANIALGORITHM(W_f, φ);
27:    Call OUTPUT(⟨key = f, value = A_f⟩);
28: end procedure
```

Figure 3.3: Pseudo-code of the second MapReduce phase of the discretization process.

CAR Mining

The generation of classification association rules is obtained via a specific adaptation of the PFP algorithm discussed in Section 3.2.1. We recall that the PFP algorithm generates

generic frequent patterns. On the other hand, the frequent patterns mined for classification are required to contain one class. This led to re-define how the FP-tree is generated from the instances and how the frequent patterns have to be mined from the FP-tree. Further, we have also proposed a sort of rule pre-pruning during the CAR mining process. The pre-pruning avoids the analysis of several sub-trees and thus the generation of several CARs. As shown in Figure 3.4, our algorithm also uses three MapReduce phases: *Parallel Counting*, *Parallel FP-Growth* and *Candidate Rule Filtering*.

Parallel Counting phase

The first MapReduce phase scans the dataset and counts the support values of each item. Each mapper analyzes an HDFS block: the input key-value pair is represented by $\langle key, value = o_i \rangle$, where o_i is an object of the training set block. For each item $v_{f,j} \in o_i$, the mapper outputs a key-value pair $\langle key = v_{f,j}, value = 1 \rangle$. The reducer is fed a list of corresponding values for each key (in this case a set of 1's) that we call *List(key)*, and outputs $\langle key = v_{f,j}, value = sum(List(key)) \rangle$. The pseudo-code of the *Parallel Counting* phase is shown in Fig. 3.5. Space and time complexity are both $O(N/Q)$, where Q is the number of CUs. Note that the *Parallel Counting* phase counts also the support of the class labels, which we assume to be the last item of each object o_i.

Only the items, called *frequent items*, whose support is larger than the support threshold $minSup$ are retained and stored in a list, called f_{list}, in descending support size order. Since f_{list} is typically small, this step can efficiently be performed on a single machine (the time complexity is $O(|f_{list}|\, log(|f_{list}|))$, where $|f_{list}|$ indicates the number of frequent items in the list). The other items are pruned and therefore not considered anymore in the subsequent phases.

Parallel FP-Growth

The second MapReduce phase, *Parallel FP-Growth*, is the core of the CAR Mining process and the relative pseudo-code is reported in Fig. 3.6. The mapper generates item-projected objects so that reducers can generate conditional FP-trees, which are independent of each other during the recursive mining process. Like in the previous phase, each mapper is fed an HDFS block and the input key-value pair is $\langle key, value = o_i \rangle$. For each o_i, the mapper gets the class label C_{l_i} and sorts the feature values according to the f_{list}. Let so_i be the sorted object. Then, for each item $so_{i,p} \in so_i$, the mapper outputs the key-value pair $\langle key = id, value = \{so_{i,1}, \ldots, so_{i,p}, C_{l_i}\} \rangle$ where id is the index of the item $so_{i,p}$ in the f_{list} and $\{so_{i,1}, \ldots, so_{i,p}, C_{l_i}\}$ is the $so_{i,p}$-projected object. Since each item is independent of the others, the reducer processes a set of independent projected objects for each single item, which represents the $v_{f,j}$-projected training set $T(v_{f,j})$. The reducer inputs a key-value pair $\langle key = id, value = T(v_{f,j}) \rangle$, builds the local FP-tree and recursively mines the classification association rules as described in [80]. Finally, it returns only the CARs whose support, confidence, and χ^2 values are greater than the relative thresholds (line 18 in Fig. 3.6). In particular, reducers output $\langle key = null, value = CAR_m \rangle$ pairs, where CAR_m is the m-th generated rule. Thus, space and time complexities of

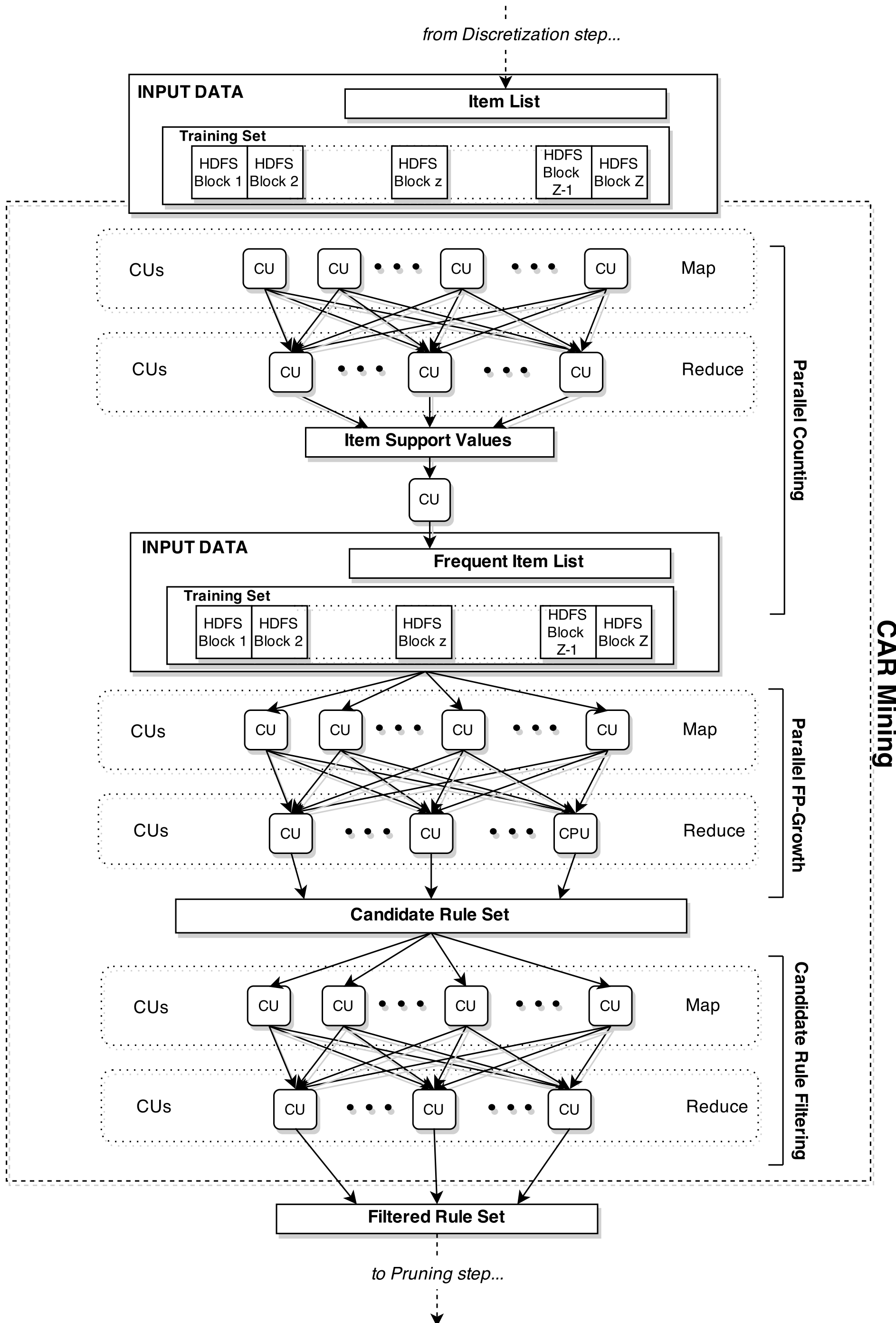


Figure 3.4: The CAR Mining step of the MapReduce Associative Classifier.

```
1: procedure MAPPER(key, value = o_i)
2:     for all item v_{f,j} in o_i do
3:         Call OUTPUT(⟨key = v_{f,j}, value = 1⟩);
4:     end for
5: end procedure
6: procedure REDUCER(key = v_{f,j}, value = List(key))
7:     sum ← 0;
8:     for all item 1 in List(key) do
9:         sum ← sum + 1;
10:    end for
11:    Call OUTPUT(⟨key = v_{f,j}, value = sum⟩);
12: end procedure
```

Figure 3.5: The MapReduce Parallel Counting Phase

each reducer depend on the size and the execution time of all the processed projected training sets, $O_{red}(Sum(|T(v_{f,j})|))$ and $O_{red}(Sum(FPGrowth(T(v_{f,j}))))$, respectively.

Figure 3.7 shows a simple example of the *Parallel FP-Growth* execution, with four objects and $minSupp = 2$. Mappers sort frequent items according to the f_{list} and create item-projected objects for each item. Reducers build the local conditional FP-tree for the specific item and recursively mine the candidate rule set. Each node on the FP-tree represents an item and each path represents an item-projected object. Since different paths share the same prefix, the tree is a compressed view of the $v_{j,i}$-projected dataset.

The recursive mining of CARs from the local FP-tree can be very time-consuming and can generate a large number of CARs. To speedup this process and reduce the number of generated CARs, we adopt a sort of rule pre-pruning during the CAR mining process. The pre-pruning avoids the inspection of several sub-trees and thus the generation of several CARs. More precisely, sub-trees that will likely generate redundant rules are not inspected. The precise explanation of the pre-pruning asks for the introduction of additional definitions. A CAR_m is more significant than another CAR_s if and only if:

1. $conf(CAR_m) > conf(CAR_s)$
2. $conf(CAR_m) = conf(CAR_s)$ **AND** $supp(CAR_m) > supp(CAR_s)$;
3. $conf(CAR_m) = conf(CAR_s)$ **AND** $supp(CAR_m) = supp(CAR_s)$ **AND** $RL(CAR_m) < RL(CAR_s)$.

where $conf(.)$, $supp(.)$ and $RL(.)$ are the confidence, the support, and the rule length, respectively. A rule $CAR_m : Ant_m \rightarrow C_{l_m}$ is more general than a rule $CAR_s : Ant_s \rightarrow C_{l_s}$, if and only if, $Ant_s \subseteq Ant_m$. Thus, a rule CAR_s can be pruned if there exists a rule CAR_m that is more significant and more general than CAR_s. This lets us discard redundant rules and cover a larger number of objects in the training set. The rules are more and more specialized as the recursive visit of a $v_{f,j}$-conditional FP-tree goes deeper in the tree. We stop the visit of an FP-tree at a specific node ι if, visiting two further nodes, the rules generated by adding the items corresponding to the nodes are not more significant than the rule generated at node ι. Let us assume that $CAR_m : Ant_m \rightarrow C_{l_m}$,

```
1: procedure MAPPER(key, value = o_i)
2:     f_list ← LOADFREQUENTITEMLIST();
3:     o_i[] ← SPLIT(o_i);                                  ▷ Array of Items
4:     C_{l_i} ← REMOVELASTITEM();
5:     so_i[] ← SORT(o_i[], f_list);       ▷ Array of Items Sorted according to f_list
6:     for p = |so_i[]| − 1 to 0 do
7:         id ← GETINDEXFLIST(so_i[p]);
8:         Call OUTPUT(⟨key = id, value = {so_i[1] . . . so_i[p], C_{l_i}}⟩);
9:     end for
10: end procedure
11: procedure REDUCER(key = id, value = T(v_{j,i}))
12:     FPTree ← NEWFPTREE();
13:     for all so_i in T(v_{j,i}) do
14:         FPTree ← INSERT(so_i);
15:     end for
16:     CAR_list ← FPGROWTH(FPTree);
17:     for all CAR_m in CAR_list do
18:         if isValid(CAR_m) then          ▷ Check support, confidence, and χ^2
19:             Call OUTPUT(⟨key = null, value = CAR_m⟩);
20:         end if
21:     end for
22: end procedure
```

Figure 3.6: The MapReduce Parallel FP-Growth Phase

$CAR_s : Ant_s \rightarrow C_{l_s}$, and $CAR_d : Ant_d \rightarrow C_{l_d}$, with $Ant_s = \{Ant_m, \widetilde{v}_{f,j}\}$ and $Ant_d = \{Ant_s, \widehat{v}_{f,j}\}$, where $\widetilde{v}_{f,j}$ and $\widehat{v}_{f,j}$ are frequent items for the Ant_s and Ant_d antecedents, respectively. We stop the generation of other rules with sub-pattern Ant_d if and only if:

$$\begin{aligned} conf(CAR_d) - conf(CAR_s) \leq 1 - \frac{supp(CAR_d)}{supp(CAR_s)} \\ AND \\ conf(CAR_s) - conf(CAR_m) \leq 1 - \frac{supp(CAR_s)}{supp(CAR_m)} \end{aligned} \tag{3.9}$$

Eq. 3.9 is not evaluated for the rules with one or two antecedents and for the rules generated from Ant_d whose class is different from CAR_d. This approach avoids the inspection of paths in the FP-tree that probably will generate redundant rules, which would be pruned in the next steps anyway. Since this modification of the FP-Growth reduces significantly the execution time of the mining process, we can reduce the value of the $minsup$ threshold and therefore extract highly confident and specialized rules with a small support. In our experiments, we verified that this approach represents a good trade-off between time and accuracy performance. Note that the number of pairs $\langle key, list(values)\rangle$ processed by each reducer is determined by the default partition function $hash(key)$ **mod** R. Since in *Parallel FP-Growth* the intermediate key is the specific item index in f_{list}, more or less the same number of pairs $\langle key, list(values)\rangle$, i.e. of conditional FP-trees, is assigned to

minSupp = 2

$f_{list} = \{v_{3,2}\ v_{2,4}\ v_{4,3}\ v_{1,5}\ v_{1,2}\}$

Parallel FP-Growth Mapper Example

Map Inputs **(o_i)** key : value = o_i	**Sorted Objects** **(so_i)** (infrequent items pruned)	**Map Outputs** **($v_{f,j}$- projected objects)** key = id : value = $so_{i,1}$, ... , $so_{i,p}$, C_{li}
$v_{1,5}\ v_{2,4}\ v_{3,2}\ v_{4,1}\ C_1$	$v_{3,2}\ v_{2,4}\ v_{1,5}$	3 : $v_{3,2}\ v_{2,4}\ v_{1,5}\ C_1$ 1 : $v_{3,2}\ v_{2,4}\ C_1$ 0 : $v_{3,2}\ C_1$
$v_{1,2}\ v_{2,2}\ v_{3,2}\ v_{4,3}\ C_1$	$v_{3,2}\ v_{4,3}\ v_{1,2}$	4 : $v_{3,2}\ v_{4,3}\ v_{1,2}\ C_1$ 2 : $v_{3,2}\ v_{4,3}\ C_1$ 0 : $v_{3,2}\ C_1$
$v_{1,2}\ v_{2,4}\ v_{3,2}\ v_{4,3}\ C_2$	$v_{3,2}\ v_{2,4}\ v_{4,3}\ v_{1,2}$	4 : $v_{3,2}\ v_{2,4}\ v_{4,3}\ v_{1,2}\ C_2$ 2 : $v_{3,2}\ v_{2,4}\ v_{4,3}\ C_2$ 1 : $v_{3,2}\ v_{2,4}\ C_2$ 0 : $v_{3,2}\ C_2$
$v_{1,5}\ v_{2,1}\ v_{3,2}\ v_{4,4}\ C_2$	$v_{3,2}\ v_{1,5}$	3 : $v_{3,2}\ v_{1,5}\ C_2$ 0 : $v_{3,2}\ C_2$

Parallel FP-Growth Reducer Example

Reduce Inputs **($v_{f,j}$- projected dataset)** key = id : value = T($v_{f,j}$)	**$v_{f,j}$- conditional FP-Tree**	**$v_{f,j}$- conditional FP-Tree** (infrequent items pruned)
0 : $\{(v_{3,2}\ C_1)\ (v_{3,2}\ C_1)\ (v_{3,2}\ C_2)\ (v_{3,2}\ C_2)\}$	$\{(C_1 : 2\ /\ C_2 : 2)\}\ \vert\ v_{3,2}$	$\{(C_1 : 2\ /\ C_2 : 2)\}\ \vert\ v_{3,2}$
1 : $\{(v_{3,2}\ v_{2,4}\ C_1)\ (v_{3,2}\ v_{2,4}\ C_2)\}$	$\{(v_{3,2} : 2\ ,\ C_1 : 1\ /\ C_2 : 1)\}\ \vert\ v_{2,4}$	$\{(v_{3,2} : 2\ ,\ C_1 : 1\ /\ C_2 : 1)\}\ \vert\ v_{2,4}$
2 : $\{(v_{3,2}\ v_{4,3}\ C_1)\ (v_{3,2}\ v_{2,4}\ v_{4,3}\ C_2)\}$	$\{(v_{3,2} : 2\ ,\ C_1 : 1\ /\ v_{2,4} : 1\ ,\ C_2 : 1)\}\ \vert\ v_{4,3}$	$\{(v_{3,2} : 2\ ,\ C_1 : 1\ /\ C_2 : 1)\}\ \vert\ v_{4,3}$
3 : $\{(v_{3,2}\ v_{2,4}\ v_{1,5}\ C_1)\ (v_{3,2}\ v_{1,5}\ C_2)\}$	$\{(v_{3,2} : 2\ ,\ v_{2,4} : 1\ ,\ C_1 : 1\ /\ C_2 : 1\ )\}\ \vert\ v_{1,5}$	$\{(v_{3,2} : 2\ ,\ C_1 : 1\ /\ C_2 : 1)\}\ \vert\ v_{1,5}$
4 : $\{(v_{3,2}\ v_{4,3}\ v_{1,2}\ C_1)\ (v_{3,2}\ v_{2,4}\ v_{4,3}\ v_{1,2}\ C_2)\}$	$\{(v_{3,2} : 2\ ,\ v_{4,3} : 1\ ,\ C_1 : 1\ /\ v_{2,4} : 1\ ,\ v_{4,3} : 1\ ,\ C_2 : 1)\}\ \vert\ v_{1,2}$	$\{(v_{3,2} : 2\ ,\ v_{4,3} : 2\ ,\ C_1 : 1\ /\ C_2 : 1)\}\ \vert\ v_{1,2}$

Figure 3.7: A simple example of the Parallel FP-Growth execution.

each reducer by the partition function. However, such a distribution does not necessarily guarantee a perfect load balancing among all the reducers, because the time spent in processing each specific conditional FP-tree depends on the number and length of its paths; more precisely, the relative time complexity is exponential with respect to the longest frequent path in the conditional pattern base [152, 212]. In case of a large number of frequent items, the conditional FP-trees corresponding to the items with the smallest support are very deep since they consider in their paths almost all the frequent items. The rule pre-pruning can reduce this problem for specific datasets, but cannot solve it in general. Thus, datasets whose objects are described by a small number of features can be easily managed, but conversely runtime problems may occur in dealing with objects with a large number of features.

Candidate Rule Filtering

The last MapReduce phase, *Candidate Rule Filtering*, selects only the K most significant non-redundant rules for each class label C_l. A rule CAR_m is not inserted into the

K most significant non-redundant rules if there exists a rule CAR_s that is more significant and more general than CAR_m. Each mapper is fed the key-value pair in the form of $\langle key = null, value = CAR_m \rangle$, and it returns a pair $\langle key = C_{l_m}, value = CAR_m \rangle$, where C_{l_m} is the CAR_m class label. Each reducer processes all the rules with the same class label, $List(CAR_{C_l})$, and selects only the K most significant non-redundant rules. For each of these K rules the reducer returns a key-value pair $\langle key = null, value = CAR_m \rangle$. Fig. 3.8 shows the pseudo-code of the *Candidate Rule Filtering* phase. We highlight that, at line 8, if the current rule CAR_m can be inserted into the K most significant non-redundant rules, the method $checkRedundant$ checks and removes all the rules that become redundant by adding the new CAR_m. Space complexity is $O(K)$ and time complexity is $O(Max(|CAR_{C_l}|) \cdot log(K)/Q)$, where Q is the number of CUs.

```
 1: procedure MAPPER(key, value = CAR_m)
 2:     C_{l_m} ← GETCLASSLABELRULE(CAR_m);
 3:     Call OUTPUT(⟨key = C_{l_m}, value = CAR_m⟩ );
 4: end procedure
 5: procedure REDUCER(key = C_l, value = List(CAR_{C_l}))
 6:     HP ← CREATEMAXHEAP(K);                        ▷ K defines the HP size
 7:     for all CAR_m in List(CAR_{C_l}) do
 8:         if checkRedundant(CAR_m, HP) then
 9:             if |HP| < K then
10:                 HP ← INSERT(CAR_m);
11:             else
12:                 if rank(HP[0]) < rank(CAR_m) then
13:                     HP ← DELETETOPELEMENT();
14:                     HP ← INSERT(CAR_m);
15:                 end if
16:             end if
17:         end if
18:     end for
19:     for all CAR_m in HP do
20:         Call OUTPUT(⟨key = null, value = CAR_m⟩);
21:     end for
22: end procedure
```

Figure 3.8: The MapReduce Candidate Rule Filtering Phase

Rule Pruning

Rule pruning aims to discard less relevant rules to speed up the classification process. Pruning has to be applied carefully, avoiding to drop useful knowledge along with discarded rules. Among the approaches proposed for rule pruning it is worth recalling lazy pruning [16], database coverage [113], and pessimistic error estimation [197].

In MRAC three different types of pruning are used. In the first type, a rule CAR_m is pruned if its support, confidence and χ^2 are not greater than *minSupp*, *minConf* and *min*χ^2 thresholds, respectively. This type of pruning is performed at the end of the *Parallel FP-Growth* phase, when the rule is mined. Since the support value is stored along the FP-tree, the computation of support, confidence, and χ^2 can be performed on the fly.

In the second type of pruning, we remove redundant rules. First, the candidate rules are sorted according to their ranking position as described in Section 3.2.2 and only the K most significant non-redundant rules for each class label are selected. Experimentally we found that this second step can reduce significantly the number of CARs in the CAR_{list}, without significantly affecting the classification accuracy. This type of pruning is performed in the new version of the FP-Growth mining algorithm, as described in Section 3.2.2, and in the reduce phase of the *Candidate Rule Filtering* job.

In the third type of pruning, shown in Fig. 3.9, the *training set coverage* is exploited: the retained rules are only those that are activated by at least one data object in the training set. Each data object in the training set is associated with a counter initialized to 0. For each object, a scan over the sorted CAR_{list} is performed to find all the rules that match the object. If CAR_m classifies correctly at least one data object, then CAR_m is inserted into the rule base. Further, the counters associated with the objects, which activate CAR_m, are incremented by 1. Whenever the counter of an object becomes larger than the *coverage threshold* δ, the data object is removed from the training set and no longer considered for subsequent rules. Since rules are sorted in descending significance, it is very likely that these subsequent rules would have a very limited relevance for the object. The procedure ends when no more objects are in the training set or all the rules have been analyzed.

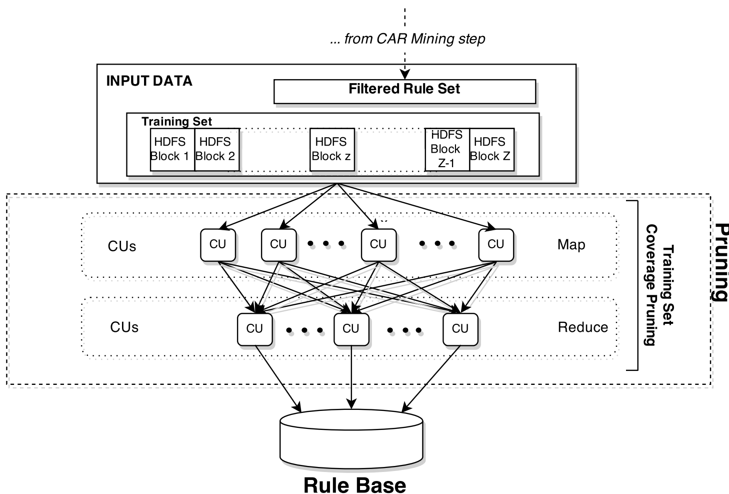


Figure 3.9: The Pruning step of the MapReduce Associative Classifier.

Fig. 3.10 shows the MapReduce pseudo-code of the third type of pruning.

```
1:  procedure MAPPER(key, value = o_i)
2:      CAR_list ← LOADANDRANKFILTEREDRULESET();
3:      δ ← LOADCOVERAGETHRESHOLD();
4:      count ← 0;
5:      for all CAR_m in CAR_list do
6:          if CAR_m matches o_i then
7:              count ← count + 1;
8:              if CAR_m correctly classifies o_i then
9:                  index_CAR_m ← GETINDEX(CAR_m, CAR_list);
10:                 Call OUTPUT(⟨key = index_CAR_m, value = null⟩);
11:             end if
12:         end if
13:         if count > δ then
14:             break;
15:         end if
16:     end for
17: end procedure
18: procedure REDUCER(key = index_CAR_m, value = null)
19:     CAR_list ← LOADANDRANKFILTEREDRULESET();
20:     CAR_m ← GETRULE(index_CAR_m, CAR_list);
21:     Call OUTPUT(⟨key = null, value = CAR_m⟩);
22: end procedure
```

Figure 3.10: The MapReduce Training Set Coverage Pruning Job

Each mapper instance loads and ranks into memory the filtered rule set, CAR_{list}, mined in the previous step. Further, since each mapper is fed an HDFS block, the key-value input pair is $\langle key = null, value = o_i \rangle$. For each object o_i, the mapper sets the counter to 0 and scans the CAR_{list}. If CAR_m matches o_i, then the counter is incremented by 1. Further, if CAR_m also correctly classifies the object o_i, the mapper outputs to the reducer the index $index_{CAR_m}$ of CAR_m in the CAR_{list} and $null$ as, respectively, key and value, that is, $\langle key = index_{CAR_m}, value = null \rangle$. When the counter exceeds the *coverage threshold* δ, o_i is not processed anymore and the next object is taken into account. The reducer instance retrieves the correct rule from the CAR_{list} and outputs it. The key-value input pair is $\langle key = index_{CAR_m}, value = null \rangle$ and the key-value output pair is $\langle key = null, value = CAR_m \rangle$. Space complexity is $O(N/Q)$ and time complexity in the worst case is $O(N \cdot |CAR_{list}|/Q)$, where $|CAR_{list}| \leq L \cdot K$ and Q is the number of CUs.

Classification

The pruned set of rules represents the *rule base* used to classify an unlabeled pattern $\widehat{\mathbf{x}}$. All rules that match the unlabeled pattern are taken into account: they can predict either

the same class label or different class labels. In the first case, MRAC simply assigns the class label to the unlabeled pattern. In the second case, the algorithm splits the rules into different groups according to the class label and compares the strength str_{C_l} of each group. The strength is computed by adopting the *weighted chi-squared* [108] as *reasoning method*:

$$str_{C_l} = \sum_{CAR_m \in RB(C_l)} \frac{\chi^2_m \chi^2_m}{max\chi^2_m} \quad (3.10)$$

where $RB(C_l)$ contains all the rules in the *rule base* with the same class label C_l, which match the unlabeled pattern, and χ^2_m and $max\chi^2_m$ are the chi-square and its upper bound for the rule CAR_m, respectively. The $max\chi^2_m$ for a generic rule $CAR_m : Ant_m \to C_{l_m}$ is calculated as follows:

$$max\chi^2_m = (minsupp(Ant_m, supp(C_{l_m})) - \frac{supp(Ant_m) \cdot supp(C_{l_m})}{N})^2 \cdot N \cdot e \quad (3.11)$$

where $supp(Ant_m)$ is the support of the antecedent of rule CAR_m, $supp(C_{l_m})$ is the support of the class label C_{l_m}, N is the number of objects in the training set and e is computed as:

$$e = \frac{1}{supp(Ant_m) \cdot supp(C_{l_m})} + \frac{1}{supp(Ant_m) \cdot (N - supp(C_{l_m}))} + \\ + \frac{1}{(N - supp(Ant_m) \cdot supp(C_{l_m})} + \frac{1}{(N - supp(Ant_m)) \cdot (N - supp(C_{l_m}))} \quad (3.12)$$

MRAC assigns the unlabeled pattern to the class label associated with the top-strength group. In case no rule matches the pattern, the method classifies $\widehat{\mathbf{x}}$ with the class label with the highest support size.

3.2.3 MRAC+: a faster version of MRAC

The algorithm described in the previous subsections is a MapReduce distributed version of the well-known CMAR algorithm, with some appropriate variation for speeding-up the execution time. Indeed, we have limited the number of items by enforcing the frequency of each interval generated in the distributed discretization process to be greater than a threshold ϕ. Then, we have modified the FP-Growth algorithm to avoid the generation of patterns that likely would yield only redundant rules. Despite such improvements, the number of rules can still be quite large (at most $K \cdot L$), thus making the pruning step performed by the database coverage very time-consuming. This number could be reduced by increasing the thresholds (confidence and support) used in the rule learning process.

From the previous analysis it becomes clear that any increase for confidence/support thresholds strongly specializes the classifier on the training set, reducing its generalization capability and therefore its accuracy on the test set. To elude this drawback,

we designed a modified version of MRAC, denoted as MRAC+ and described hereafter. MRAC+ do not use the training set coverage and adopts the *best rule* method as inference mechanism. Given an unlabeled pattern $\widehat{\mathbf{x}}$, we output the class corresponding to the first activated rule in the sorted rule base obtained after applying the first two types of pruning described in Section 3.2.2. Thus, we assign to $\widehat{\mathbf{x}}$ the class associated with the most confident and specialized rule in the rule base. In Section 3.2.4, we highlight that MRAC+ outperforms MRAC in terms of both accuracy and computation time over the big datasets used in our experimental study. For the sake of clarity, Figure 3.11 summarizes the workflow of MRAC and MRAC+ for both the learning and the classification processes.

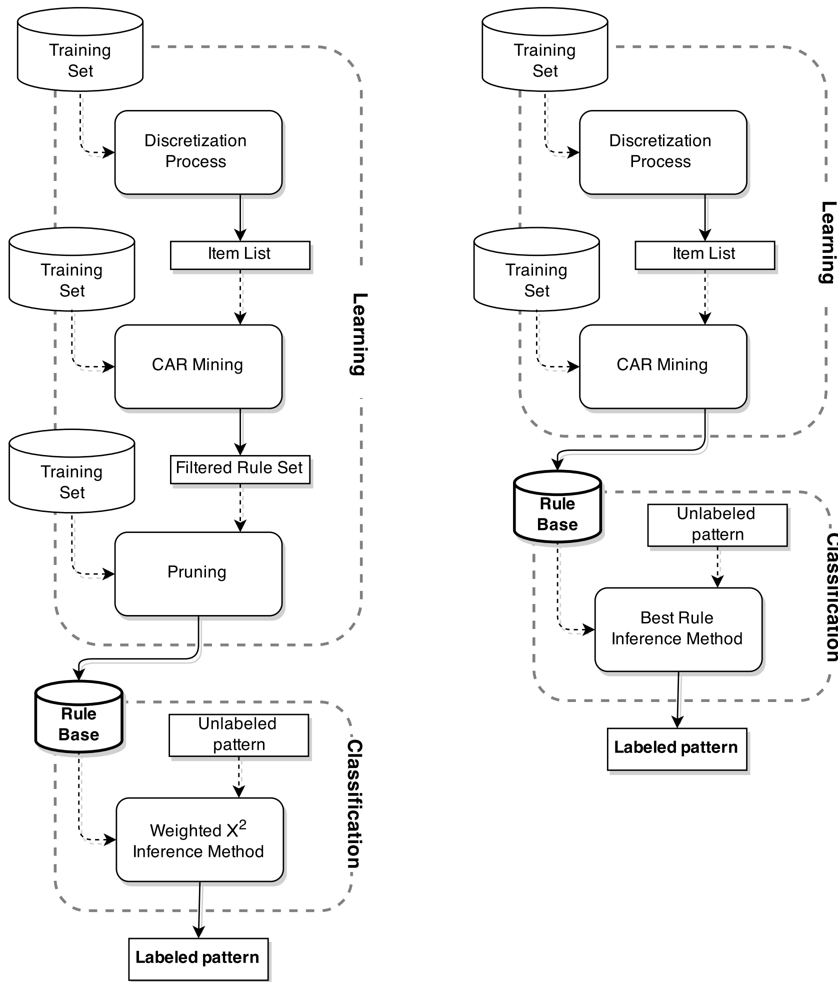


Figure 3.11: The overall workflow of MRAC (on the left) and MRAC+ (on the right).

3.2.4 Experimental Study

Specific experimental tests have been devised to characterize the behavior of the proposed algorithms, focusing on the following crucial aspects: i) performance in terms of classification accuracy, model complexity and computation time, ii) horizontal scalability analysis with a typical complete dataset, and iii) study on the ability to efficiently accommodate an increasing dataset size.

As shown in Table 3.1, we employed 7 well-known big datasets, extracted from the UCI repository[1] and LIBSVM repository[2], which are characterized by different numbers of input/output instances (from 581012 to 11000000), classes (from 2 to 23), and attributes (from 10 to 54). For each dataset, we report the number of numeric (N) and categorical (C) attributes.

Table 3.1: Big datasets used in the experiments.

Dataset	# Instances	# Attributes	# Classes
Cover Type (COV)	581012	54 (10 N, 44 C)	2
HIGGS (HIG)	11000000	28 (28 N)	2
KDDCup 1999 2 Classes (KDD99_2)	4856151	41 (26N, 15C)	2
KDDCup 1999 5 Classes (KDD99_5)	4898431	41 (26N, 15C)	5
KDDCup 1999 (KDD99)	4898431	41 (26N, 15C)	23
Poker-Hand (POK)	1025010	10 (10 C)	10
Susy (SUS)	5000000	18 (18 N)	2

All the experiments have been run using Apache Hadoop 1.0.4 as the reference MapReduce implementation. The chosen testbed corresponds to a typical low-end system suitable for supporting the target classification service: a small cluster with one master and three slave nodes, connected by a Gigabit Ethernet (1 Gbps). All the nodes run Ubuntu 12.04. Regarding the deployment of the Hadoop components, the master hosts the *NameNode* and *JobTracker* processes, while each slave runs a *DataNode* and a *TaskTracker*. The *NameNode* is devoted to handling the HDFS, keeping track of its block replicas (64 MB by default), and to coordinating all the *DataNode* processes as well. The master node has a 4-core CPU (Intel Core i5 CPU 750 x 2.67 GHz), 4 GB of RAM and a 500GB Hard Drive. Each slave node has a 4-core CPU with Hyperthreading (Intel Core i7-2600K CPU x 3.40 GHz), 16GB of RAM and 1 TB Hard Drive.

Given the practical importance of time efficiency in several data analysis tasks, we have also carried out preliminary investigations on possible improvements from Spark-based versions of the proposed classification scheme. To this aim, we have compared the version of the PFP algorithm, which has been recently implemented in the MLlib library on Spark (version 1.4.1), with the Mahout version used for our algorithm. Surprisingly, we have verified that the MLlib version outperforms the Mahout one only when the

[1] Available at Available at `https://archive.ics.uci.edu/ml/datasets.html`
[2] Available at Available at `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/`

number of frequent items is small. When the number of frequent items is large, the Mahout version behaves slightly better. This behavior is mainly due to the lack in the MLlib implementation of the aggregating step suggested in [106] and developed in the Mahout version. This means that a really effective implementation on Spark of the overall proposed classification scheme will be possible only upon the availability of the overall PFP proposed in [106] in the MLlib library.

Performance of MRAC+ and MRAC

In this section, we analyze the performance of MRAC+ and MRAC in terms of accuracy, model complexity, and computation time. To the best of our knowledge, no other implementation of associative classifiers has been proposed in the literature for handling big datasets. Thus, to assess the performance of MRAC and MRAC+ in comparison with other algorithms, we have employed well-known recent distributed implementations of two different classifier types, namely Decision Tree (DT) and Random Forest (RF).

As regards DT, we used the implementation available in MLlib [140] that performs a recursive binary partitioning of the feature space. For the generation of partitions (at most *maxBins*), the algorithm computes a set of split candidates by performing a quantile calculation over a sampled fraction of the data. Then, at each decision node, each partition is chosen greedily by selecting the best split from the set of possible splits, in order to maximize the information gain, measured as node *impurity*. The maximum possible depth *maxDepth* of the tree can be fixed by the user: deeper trees are more expressive (potentially allowing higher accuracy on the training set), but they also ask for a longer learning process and are characterized by a higher probability of overtraining.

As regards RF [23], we used the implementation available in Mahout [127]. An RF uses bagging in tandem with random attribute selection for generating a multitude of decision trees and outputs the class that is the mode of the classes of the individual trees. At each node of each tree of the forest, a subset G of the available attributes is randomly chosen and the best split available within these attributes is selected for that node. To deal with big datasets, the Mahout algorithm uses a partial implementation that builds multiple trees for different blocks of data. First, a partitioning of the dataset into independent data blocks of dimension $maxSplitSize$ is performed. Then, each Mapper is fed by one data block and builds a subset of the random forest. The set of trees generated by each Mapper forms the forest.

Table 3.2 summarizes, for each algorithm, the parameters used in the experiments. For DT and RF, we adopted the values suggested in the guidelines provided with the libraries. As regard the $maxBins$ parameter of the DT and the KDDCup datasets, we raise up the value to 70, since for the categorical attributes it has to be at least the same number of the possible values of feature. For each dataset and for each algorithm, we performed a five-fold cross-validation by using the same folds for all the datasets.

Table 3.3 shows, for each dataset and for each algorithm, the average values $\pm$ standard deviation of the accuracy, both on the training (Acc_{Tr}) and test sets (Acc_{Ts}) obtained by the four algorithms. The highest accuracy values for each dataset are shown

Table 3.2: Values of the parameters for each algorithm used in the experiments.

Method	Parameters
MRAC+	$\gamma = 0.1\%, \phi = 2\%, MinSupp = 0.01\%, MinConf = 50\%, min\chi^2 = 20\%, K = 15000$
MRAC	$\gamma = 0.1\%, \phi = 2\%, MinSupp = 0.01\%, MinConf = 50\%, min\chi^2 = 20\%, K = 15000, \delta = 4$
Decision Tree	$MaxDepth = 5, maxBins = 32, Impurity = GINI$
Random Forest	$NumTrees = 100, G = \lceil log_2 F \rceil, maxSplitSize = 64MB$

in bold. Table 3.4 summarizes the computation times (in seconds) spent by each algorithm on a cluster of 3 slaves with 4 cores per slave (12 cores in total). For the Hadoop configuration, the number of mappers and reducers is set equal to the available cores on the cluster. Moreover, we report also the number of HDFS blocks (Z) and instances per block (N_{Block}). As regards MRAC+ and MRAC, it is worth noting that due to memory constraint, for the HIG dataset, we set the number of reducers to 2. Moreover, for the RF and poker-hand datasets, because of limitations due to the computational costs, we drop down the value of the $maxSplitSize$ parameter to 4.5MB, so that each mapper computes a subset of the random forest on a quarter of the overall training set (200,002 instances per block). It is worth mentioning that, by using the default configuration, we did not obtain any result after 26 hours. The analysis of the two tables highlights that, on average, MRAC+ outperforms MRAC in terms of accuracy on two datasets, and it is comparable to MRAC in the other five datasets. Furthermore, the learning process is faster in MRAC+ than in MRAC of one order of magnitude. Thus, we can conclude that MRAC+ is certainly more effective than MRAC for the big datasets used in our experiments. As regards DT, we note that MRAC+ achieves higher classification rates than DT on the COV, POK and SUS datasets: on the other datasets, the classification rates are comparable. MRAC achieves higher classification rates than DT only on the POK dataset: on the other datasets, the classification rates are comparable. On the POK dataset, we observe that both MRAC+ and MRAC obtain classification rates much higher than DT and higher than RF. This result is due to a specificity of the POK dataset, which contains only categorical attributes. For this type of datasets, the associative classifiers perform particularly well. This conclusion is supported also by the analysis of the results on the COV dataset, where the number of categorical attributes is higher than numeric attributes (44 against 10). Also in this case, the associative classifiers outperform the other two types of classifiers. Table 3.4 shows that DT is much faster than the other comparison algorithms. On the other hand, we have to consider that MRAC+, MRAC, and RF are implemented on Hadoop, while DT is implemented on Spark, which has been designed to be more efficient in managing data reading/writing than Hadoop [206]. Thus, this runtime difference is mainly due to the execution environment rather than to the complexity of the algorithms. As regards RF, MRAC+ and MRAC outperform RF on the COV and POK datasets, as already observed. RF achieves higher accuracies than MRAC+ and MRAC on the HIG and SUS datasets, which are the two datasets with only numeric attributes. On the three KDD datasets, the classification rates of RF are slightly higher than MRAC+ and MRAC. Table 3.4 shows that the computation time of RF is of the same order of magnitude of

MRAC+ and MRAC. On the other hand, RF is implemented on Hadoop as MRAC+ and MRAC.

Table 3.3: Average accuracy $\pm$ standard deviation achieved by MRAC+, MRAC, Decision Tree (DT), and Random Forest (RF).

	MRAC+		**MRAC**		**DT**		**RF**	
Dataset	Acc_{Tr}	Acc_{Ts}	Acc_{Tr}	Acc_{Ts}	Acc_{Tr}	Acc_{Ts}	Acc_{Tr}	Acc_{Ts}
COV	$\mathbf{78.329} \pm 0.091$	$\mathbf{78.092} \pm 0.157$	74.246 ± 0.100	74.261 ± 0.156	74.148 ± 0.199	74.140 ± 0.173	70.198 ± 0.868	70.068 ± 0.837
HIG	65.942 ± 0.058	65.904 ± 0.045	65.079 ± 0.054	65.050 ± 0.061	66.376 ± 0.074	66.375 ± 0.058	$\mathbf{73.006} \pm 0.016$	$\mathbf{72.542} \pm 0.025$
KDD99_2	$\mathbf{99.999} \pm 0.000$	99.998 ± 0.000	99.959 ± 0.002	99.957 ± 0.004	99.978 ± 0.014	99.979 ± 0.013	$\mathbf{99.999} \pm 0.000$	$\mathbf{99.999} \pm 0.000$
KDD99_5	99.863 ± 0.046	99.858 ± 0.047	99.898 ± 0.034	99.898 ± 0.035	99.776 ± 0.064	99.775 ± 0.063	$\mathbf{99.986} \pm 0.002$	$\mathbf{99.982} \pm 0.002$
KDD99	99.582 ± 0.020	99.579 ± 0.020	99.640 ± 0.024	99.639 ± 0.024	99.781 ± 0.059	99.782 ± 0.057	$\mathbf{99.968} \pm 0.006$	$\mathbf{99.966} \pm 0.006$
POK	$\mathbf{94.480} \pm 0.000$	$\mathbf{94.480} \pm 0.000$	$\mathbf{94.480} \pm 0.000$	$\mathbf{94.480} \pm 0.000$	55.165 ± 0.213	55.191 ± 0.203	91.277 ± 0.127	89.591 ± 0.287
SUS	78.247 ± 0.013	78.220 ± 0.035	76.245 ± 0.055	76.232 ± 0.068	77.119 ± 0.040	77.118 ± 0.046	$\mathbf{80.671} \pm 0.009$	$\mathbf{80.064} \pm 0.033$

Table 3.4: The computation times (in seconds) for the learning process in MRAC+, MRAC, Decision Tree (DT), and Random Forest (RF).

Dataset	$\mathbf{Z}$	$\mathbf{N_{Block}}$	**MRAC+**	**MRAC**	**DT**	**RF**
COV	1	464,809	504	1059	16	459
HIG	96	91,667	6141	9881	289	196
KDD99_2	6	47,487	669	704	23	246
KDD99_5	6	653,124	1439	1708	29	258
KDD99	6	653,124	1878	2280	28	263
POK	1	800,008	239	1099	10	2212
SUS	29	137,932	738	3713	106	865

For the sake of completeness, we mention that the classification rates of MRAC+ and MRAC are also higher than the ones reported in [175]. Here, the authors investigate several prototype reduction techniques on Hadoop with the aim of improving the classification rates of the nearest neighbor classifier. These methods have proven to be very competitive in reducing the computational cost and high storage requirements of the nearest neighbor classifier, improving its classification performance. Since the authors adopt only three datasets, we have not shown the results in Table 3.3. We highlight however that their best solutions achieve an average accuracy of 99.94%, 51.81% and 72.82% on the test set for the KDD99_2, POK and SUS datasets, respectively.

Table 3.5 shows the complexity of each algorithm. For MRAC+ and MRAC we report the numbers of frequent items ($|f_{list}|$) and of rules ($\#Rules$). Obviously the size of f_{list} is equal on both algorithms since the discretization process and the *Parallel Counting* job is the same for both approaches. For MRAC, we also report the percentage of rules discarded ($Pruning(\%)$) by the database coverage step. As regards DT and RF, we report the number of nodes ($\#Nodes$) of the tree for DT and the average maximum depth ($maxDepth$) achieved by the forest for RF.

As shown in Table 3.5, both MRAC+ and MRAC turn to be not very interpretable. Indeed, in both classifiers the number of rules is very high, even if in MRAC the use of the

Table 3.5: Complexities of MRAC+, MRAC, Decision Tree (DT), and Random Forest (RF).

		MRAC+	MRAC		DT	RF	
Dataset	$\lvert f_{list} \rvert$	$\#Rules$	$\#Rules$	$Pruning(\%)$	$\#Nodes$	$\#Nodes$	$maxDepth$
COV	155	15612	6714	57.00	63	46774	17
HIG	319	29999	19468	35.10	63	1807880	38
KDD99_2	189	30000	1174	96.09	35	27545	10
KDD99_5	201	49349	2878	94.17	45	57102	12
KDD99	203	125294	2806	97.76	46	56691	11
POK	85	5980	5353	10.48	63	6166485	7
SUS	345	30000	21963	26.79	63	2106179	47

database coverage step reduces it significantly. Database coverage tends to discard the rules that are at the bottom of the ranked rule list: when rules generated at the end of the *CAR Mining* step are characterized by a large support, such as for KDD99_2, KDD99_5, and KDD99 datasets, the pruning effects are more evident. As regards MRAC+, since the activated rules do not contain, in general, a large number of conditions (at most four or five on average), we can affirm that the classifier can provide a very interpretable explanation for each conclusion (consider that we consider just a rule in the inference process). Thus, even if the number of rules is large in the final rule base, we can consider that, for each unlabeled pattern, the classifier can provide a very intuitive justification of its reasoning. As regards the other algorithms, RF is characterized by a very high number of nodes, thus having a higher level of complexity than MRAC+, MRAC, and DT. DT has a much lower number of nodes than RF and is certainly the least complex algorithm.

Scalability analysis

In this section, we investigate the MRAC+ and MRAC behaviors in employing additional computing units. To this aim, we measure the values assumed by the *speedup* σ, taken as the main metrics, commonly used in parallel computing. For the sake of clarity, we report the figures obtained with tests on the Susy dataset; similar results can be recorded with the other datasets.

According to the speedup definition, the efficiency of a program using multiple CUs is calculated comparing the execution time of the parallel implementation against the corresponding sequential, “basic” version. In our application setting, because of the large size of the involved datasets, it is not practically sensible to regard the sequential version of the overall algorithm as the basic one (it would take an unreasonable amount of time), so we can refer to a run over Q^* identical CUs, $Q^* > 1$. Hence, we adopt the following slightly different definition for the speedup on n identical CUs:

$$\sigma_{Q^*}(n) = \frac{Q^* \cdot \tau(Q^*)}{\tau(n)} \tag{3.13}$$

where $\tau(n)$ is the program runtime using n CUs, and Q^* is the number of CUs used to run the reference execution, which lets us estimate a fictitious, ideal single-core runtime

as $Q^* \cdot \tau(Q^*)$. Of course, $\sigma_{Q^*}(n)$ makes sense only for $n \geq Q^*$. In our case, $\tau(Q^*)$ accounts also for the basic overhead due to the Hadoop platform.

For $n > Q^*$ the speedup is expected to be sub-linear because of the increasing overhead from the Hadoop procedures, because of the behavior of the algorithm (considering also the granularity of the necessary sequential parts) and, in case of multicore physical nodes, because of contention on resources shared among cores within the same CPU.

In our tests, we assumed $Q^* = 6$ to have 2 working cores for each slave available in the cluster and thus accounting in σ_6 also for the basic overhead due to thread interference.

Considering the structure of our algorithm, we set the number of reducers equal to the number of cores and we distribute them uniformly among the slaves.

Horizontal scalability has been studied by varying the number of switched-on cores per node. To avoid unbalanced loads, we recorded the execution times experienced with the same number of running cores per node. In practice, we considered 6, 9, and 12 cores distributed on the three slave nodes.

It is worth noticing that the HyperThreading technology was available on our testbed CPUs, which thus might run two distinct processes per core. The performance gain due to HypertThreading highly depends on the target application, and in server benchmarks it reaches 30% [130]. In our case, specific tests showed that HyperThreading yields really limited performance improvements. For this reason we disabled the HyperThreading Technology and used only the available physical CUs in all our experiments.

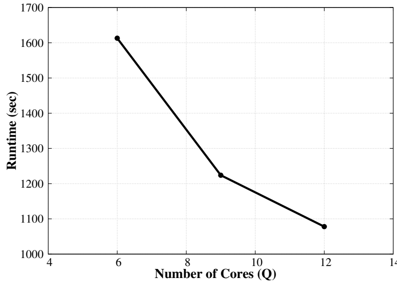
Table 3.6 and Figure 3.12 show the speedup according to the whole dataset for MRAC+ and MRAC.

Table 3.6: Runtime, speedup (σ_6), and utilization ($\sigma_6(Q)/Q$) for MRAC+ and MRAC on the Susy dataset.

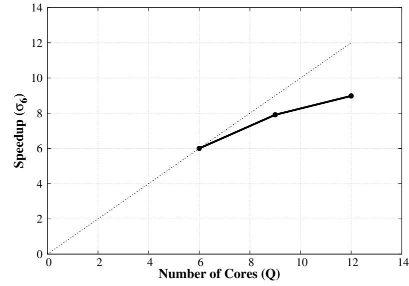
	MRAC+			**MRAC**		
# Cores	**Time (s)**	$\sigma_6(Q)$	$\sigma_6(Q)/Q$	**Time (s)**	$\sigma_6(Q)$	$\sigma_6(Q)/Q$
6	1613	6	1.00	7944	6	1.00
9	1224	7.91	0.88	5642	8.45	0.94
12	1006	9.62	0.80	4412	10.80	0.90

With the default Hadoop settings, the number Z of mappers is automatically determined by the HDFS block size. E.g., for the Susy dataset Hadoop instantiates 36 mappers. Furthermore, indicating by Q the number of available cores, if $Z \leq Q$ then all the mappers are run simultaneously, and the global runtime practically corresponds to the longest of the mappers' runtimes. Otherwise ($Z > Q$), Hadoop starts from executing Q mappers in parallel, queuing the rest $(Z - Q)$. As soon as one of the running mappers completes, Hadoop schedules a new mapper from the queue.

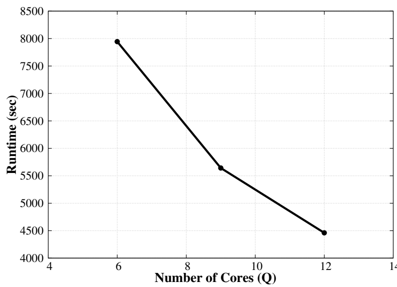
In the ideal case of the same execution time for all the mappers, the map phase for each MapReduce stage would require $\lceil \frac{Z}{Q} \rceil$ iterations. With the Susy dataset, this



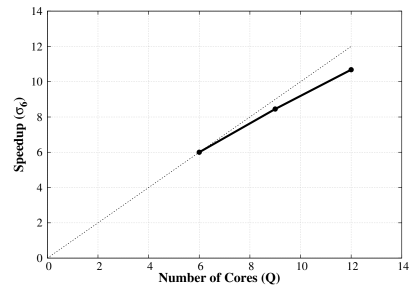
(a) Runtime of MRAC+



(b) Speedup of MRAC+



(c) Runtime of MRAC



(d) Speedup of MRAC

Figure 3.12: Runtime and Speedup for MRAC+ (a-b) and MRAC (c-d) on the overall Susy dataset.

corresponds to 6, 4 and 3 iterations on 6, 9 and 12 cores, respectively. This observation can be used to get a very rough estimation of the runtime expected with a certain number of cores, once the runtime with another given number of cores has been recorded. Such an estimation cannot be accurate because all the mappers do not have exactly the same execution time (different input sizes may even be assigned to them) and because of the influence of the different reducing phases. For instance (see Table 3.6), we expect that the MRAC+ runtime would decrease from 1613 seconds with 6 cores to about $1613 \times 4 \div 6 = 1076$ and $1613 \times 3 \div 6 = 807$ seconds with 9 and 12 cores, respectively. Similarly for MRAC, the runtime should decrease from 7944 seconds with 6 cores to about $7944 \times 4 \div 6 = 5296$ and $7944 \times 3 \div 6 = 3972$ seconds with 9 and 12 cores, respectively. As it can be noticed, such values do not excessively differ from the measured ones. Of course, the actual runtimes are necessarily higher due to the incurred overheads.

The actual speedup σ_6 in our experiments shows a different behavior depending on the algorithm. As regard MRAC, σ_6 does not excessively diverge from the ideal value,

i.e. the number of CUs[3]: $\sigma_6(9)/9 = 0.94$ and $\sigma_6(12)/12 = 0.90$. On the other hand, the MRAC+ speedup rapidly decreases: $\sigma_6(9)/9 = 0.88$ and $\sigma_6(12)/12 = 0.80$. However, within the limitations due to the different experimental settings, this result is in line with [106] where the PFP was introduced and the utilization is 0.768. It can be noted also that the contribution of the PFP algorithm on the overall learning process is much more relevant in MRAC+ than in MRAC.

For a better understanding of the different speedups of the two algorithms, a breakdown of the contributions from the different parts is required. To this aim, in Table 3.7 and Figure 3.13 we report the speedup σ_6 of the discretization process and the most significant MapReduce phases of the *learning* step: *Parallel FP-Growth* and *Parallel Training Set Coverage Pruning*. The contribution of the other two, i.e. *Parallel Counting* and *Candidate Rule Filtering* is negligible (about 2.46% and 1.08% of the overall execution time, respectively for MRAC).

Table 3.7: Runtime, speedup (σ_6), and utilization ($\sigma_6(Q)/Q$) of the *Discretization*, *Parallel FP-Growth*, and *Training Set Coverage Pruning* phases in the Susy dataset.

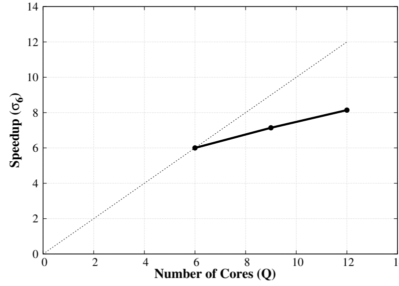
	Discretization			**Parallel FP-Growth**			**Training Set Coverage Pruning**		
# Cores	**Time (s)**	$\sigma_6(Q)$	$\sigma_6(Q)/Q$	**Time (s)**	$\sigma_6(Q)$	$\sigma_6(Q)/Q$	**Time (s)**	$\sigma_6(Q)$	$\sigma_6(Q)/Q$
6	175	6	1.00	1190	6	1.00	6329	6	1.00
9	147	7.14	0.79	873	8.18	0.91	4418	8.60	0.96
12	125	8.40	0.70	701	10.19	0.85	3405	11.15	0.93

The three charts in Figure 3.13 clearly show that the three phases behave differently with respect to scalability.

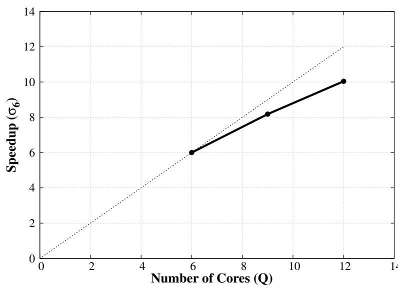
The speedup of the *Discretization* phase (Figure 3.13a) rapidly degrades, mostly because the computational cost mainly depends on the number of attributes rather than on the number of HDFS blocks. Indeed, the overall execution time of the *Discretization* phase is affected by the computation of the equi-frequency bins, performed in the mapping phase, and the Fayyad and Irani algorithm, executed in the reducing phase. With the Hadoop default settings, the attributes are evenly distributed among the reducers. In our tests, the Susy dataset has 18 continuous attributes, thus each reducer processes 3, 2 and 2 attributes in case of 6, 9 and 12 cores, respectively. Note that since with 12 cores only 6 reducers handle 2 attributes, the computational weight is not evenly balanced within the cluster. Thus, by adding cores we can improve the discretization runtime, decreasing the number of attributes processed by each reducer.

As regards the *Parallel FP-Growth* phase (Figure 3.13b), the main contribution to the computational weight is due to the reducing activity, but however the speedup shows a better trend than in the *Discretization* phase. The average runtime of each mapper is quite short (about 40 seconds), and the global runtime is dominated by the reducing phase,

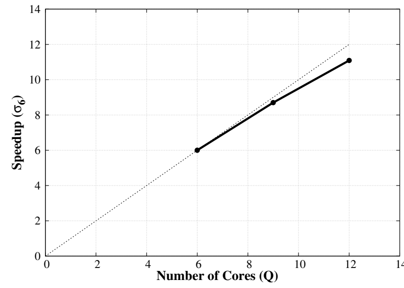
[3] The value σ_1/Q is the standard *utilization* index; in our case, as $\sigma_i(n) \leq \sigma_j(n)\ \forall n \geq j$, the utilization index σ_6/Q may be slightly greater than standard utilization.



(a) *Discretization* speedup.



(b) *Parallel FP-Growth* speedup.



(c) *Training Set Coverage Pruning* speedup.

Figure 3.13: Speedup of the discretization and the two main *learning* phases

where CARs are mined out of the conditional FP-trees. With the Hadoop default settings, all the conditional FP-trees are evenly distributed among the reducers.

In our tests, Susy has 353 frequent items, thus each reducer processes about 59, 40 and 30 conditional FP-trees in the 6, 9 and 12 cores cases, respectively. In *Parallel FP-Growth*, adding more cores helps in improving the FP-Growth parallelization, by decreasing the number of conditional FP-trees processed by each reducer.

Conversely, the *Training Set Coverage Pruning* is driven by the map phase, with very satisfactory utilization values. In this case, the average runtime of each mapper is around 18 minutes. The global execution time can be shrunk by reducing the number of iterations, i.e. by exploiting additional CUs, as witnessed by the results in Table 3.7.

Tackling the dataset size

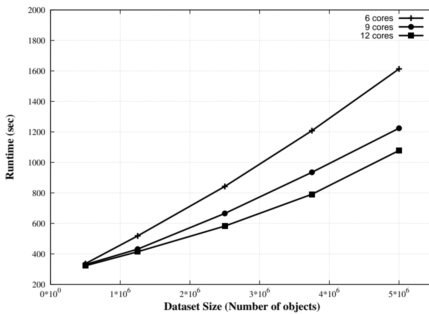
From a practical point of view, it is crucial to understand how the proposed algorithm behaves as the input dataset size grows up. To test this aspect, we extracted differently sized datasets out of Susy. For each given size, three different experiments have been executed over three distinct subsets of Susy, with records randomly sampled out of the

complete dataset. We indicate a subset with $x\%$ of the records in Susy by the notation $Susy_x$; thus, the complete dataset is $Susy_{100}$.

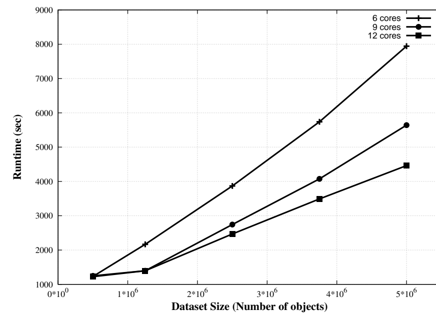
Table 3.8 and Figure 3.14 show the average runtime for building the rule base, according to different problem sizes and number of cores.

Table 3.8: Average runtime of MRAC+ and MRAC on the Susy dataset, varying the dataset size and the number of available cores.

Dataset			MRAC+			MRAC		
Size (%)	**Objects**	**Mappers**	**6**	**9**	**12**	**6**	**9**	**12**
10 ($Susy_{10}$)	500,000	4	336	330	322	1230	1247	1224
25 ($Susy_{25}$)	1,250,000	9	518	431	414	2167	1396	1397
50 ($Susy_{50}$)	2,500,000	18	842	665	582	3869	2743	2468
75 ($Susy_{75}$)	3,750,000	27	1208	935	791	5738	4075	3490
100 ($Susy_{100}$)	5,000,000	36	1613	1224	1006	7944	5642	4412



(a) Average Runtime of MRAC+ on Susy dataset.



(b) Average Runtime of MRAC on Susy dataset.

Figure 3.14: Average Runtime of both MRAC+ (a) and MRAC (b) on the Susy dataset, varying the dataset size and the number of available cores.

As shown for $Susy_{10}$ and $Susy_{25}$, whenever the number of available cores Q is sufficient to run all the mappers in parallel, adding more cores does not trivially yield any benefit. When the number of mappers exceeds the number of cores, Hadoop queues up the extra mappers to be subsequently scheduled when cores become available again. Thus, as long as the number of mapper iterations is the same, no significant gain is expected by adding new cores. For instance, for $Susy_{50}$ passing from 9 to 12 cores is of little practical use, since the number of parallel mapper iterations (namely, two) does not change. In this case, runtime improves only of 83 and 275 seconds for MRAC+ and MRAC, respectively. On the other hand, the runtime difference passing from 6 to 9 cores

is more significant (about 177 and 1126 seconds for MRAC+ and MRAC, respectively), because a reduction of the number of parallel mapper iterations occurs (from 3 to 2). Similar considerations can be made for $Susy_{75}$ and $Susy_{100}$. Note that in the last case, passing from 6 to 9 cores determines a reduction of iterations from 6 to 4, while by adding yet other three cores the iterations just go down to 3, i.e. yielding half of the previous gain.

Experiments show that, as the dataset size grows up, the execution time can be effectively reduced by adding a proper number of additional cores, at least whenever dealing with sizes typical of current big data benchmarks. In particular, performance improvements are mainly related to the reduction in the number of parallel mapper iterations spent by the algorithm to scan the overall dataset.

3.3 Fuzzy Associative Classifiers

As described in Section 3.1, association rule mining has become a very popular method to build highly accurate classification models. Such method is able to mine a set of high quality classification rules from huge amounts of data and to achieve a considerable performance in terms of classification accuracy. However, as stated in [149], even though learning based on association rule mining ensures high accuracy in pattern classification and generates rule-based models that are often “interpretable” by the user, this model suffers from some main weaknesses. First, the complexity of the learning process grows exponentially in terms of both time and memory with the number of training data objects. Second, association rule mining algorithms deal with binary or categorical itemsets, but real data objects are often described by numerical continuous features. Thus, appropriate discretization algorithms have to be applied to transform continuous feature domains into a set of items.

However, as regards the complexity of the learning process, MRAC and his enhanced version MRAC+ have proved to build accurate models in a reasonable time, achieving comparable results in terms of classification rate and computational time with the ones obtained by other algorithms. Thus, MapReduce or more general distributed approaches can be employed to handle a huge amount of data. On the other hand, rgarding the issue of managing continuous input variables, associative classification approaches as well as MRAC described in Section 3.2.2 adopt discretization algorithms for extracting a set of items and therefore for allowing the rule mining algorithms to work properly. The discretization is accomplished by assigning each value to a bin. The data ranges (*bin boundaries*) and the number of bins are determined by the discretization algorithm. Bin boundaries are typically crisp, but crisp discretization is not natural. Indeed, the transitions between bins are not generally abrupt, but rather gradual. Thus, fuzzy sets are certainly more appropriate for describing attribute partitions. In the last years, several studies and different algorithms have been proposed to integrate associative classification models with the fuzzy set theory, leading to the so-called *fuzzy associative classifiers*. All these associative classification approaches have used fuzzy boundaries, thus generating fuzzy association rules [6, 34, 63, 123, 125, 147, 149]. As stated in [123, 125], the use of fuzzy

association rules can restrain the sharp boundary effect between intervals, because fuzzy set concepts provide a smooth transition between intervals, resulting in fewer boundary elements being excluded.

In [34], authors introduce a fuzzy associative classifier based on Apriori to mine all fuzzy CARs: notions of support, confidence, redundancy and rule conflict have been extended to the fuzzy context for selecting only the best CARs to build the classifier. Similarly, in [149], authors propose an associative classification model, which generates fuzzy CARs by means of a fuzzy version of Apriori. Different methods for generating the initial fuzzy partitions and for classifying the patterns have been experimented.

Also some recent works [6, 63] exploit the Apriori algorithm for mining fuzzy CARs. The Fuzzy Association Rule-based Classification model for High Dimensional datasets (FARC-HD), proposed in [6], is a fuzzy associative classification approach consisting of three steps. First, all possible fuzzy association rules are mined by applying the Apriori algorithm, limiting the cardinality of the itemsets, so as to generate fuzzy rules with a low number of conditions. Then, a pattern weighting scheme is employed to reduce the number of candidate rules, pre-selecting the most interesting. Finally, a single objective evolutionary algorithm is applied to select a compact set of fuzzy association rules and to tune the membership functions. In [63], the authors discuss the D-MOFARC algorithm, which extends the FARC-HD algorithm to the multi-objective context. In particular, unlike in FARC-HD, a multi-objective evolutionary algorithm has been employed for the post-processing stage. Moreover, a tree-based generation mechanism for generating initial fuzzy partitions has been integrated. This mechanism is based on the recursive application of the CAIM discretization algorithm [103] for taking attribute partitioning interdependencies into consideration.

The most recent papers regarding fuzzy associative classifiers mainly focus on the application of these models to specific domains, such as recommender systems [123, 163], decision making [147], predicting the growth of sellers in an electronic marketplace [125], dealing with imprecise data [151]. Most of these papers propose fuzzy associative classifiers that are extensions to the fuzzy context of non-fuzzy associative classifiers previously proposed in the literature. For example, in [123], the authors introduce a fuzzy version of the well-known CBA algorithm [113] and discuss an example of how associative classification models can be used for building recommender systems. In [147] the evolution of a learning classifier system, designed to extract quantitative association rules from unlabeled data streams, is described. The proposed system evolves a population of fuzzy association rules. At the end of the learning process, the population is expected to contain rules that capture the most interesting associations between problem attributes. Finally, the work in [125] presents a fuzzy extension of a GARC classifier, which includes also a method for learning the initial fuzzy partitions employing the simulated annealing optimization algorithm.

3.3.1 Fuzzy Rule Based Classifiers

Pattern classification consists of assigning a class C_l from a predefined set $C = \{C_1, \ldots, C_L\}$ of classes to an unlabeled pattern. We consider a pattern as an F-dimensional point in a feature space $\Re^F$. Let **X** $= \{X_1, \ldots, X_F\}$ be the set of input variables and U_f, $f = 1, \ldots, F$, be the universe of discourse of the fth variable. Let $P_f = \{A_{f,1}, \ldots, A_{f,T_f}\}$ be a fuzzy partition of T_f fuzzy sets on variable X_f. The data base (DB) of an FRBC is the set of parameters which describe the partitions P_f of each input variable. The rule base (RB) contains a set of M rules usually expressed as:

$$R_m : \textbf{IF } X_1 \textbf{ is } A_{1,j_{m,1}} \textbf{ AND} \ldots \textbf{AND } X_F \textbf{ is } A_{F,j_{m,F}} \\ \textbf{THEN } Y \textbf{ is } C_{j_m} \; with \; RW_m \tag{3.14}$$

where Y is the classifier output, C_{j_m} is the class label associated with the mth rule, $j_{m,f} \in [1, T_f]$, $f = 1, ..., F$, identifies the index of the fuzzy set (among the T_f linguistic terms of partition P_f), which has been selected for X_f in rule R_m. RW_m is the rule weight, i.e., a certainty degree of the classification in the class C_{j_m} for a pattern belonging to the fuzzy subspace delimited by the antecedent of rule R_m.

Let $(\mathbf{x}_n, y_n)$ be the nth input-output pair, with $\mathbf{x}_n = [x_{n,1} \ldots, x_{n,F}] \in \Re^F$ and $y_n \in C$. The strength of activation (*matching degree* of the rule with the input) of the rule R_m is calculated as:

$$w_m(\mathbf{x_n}) = \prod_{f=1}^{F} A_{f,j_{m,f}}(x_{n,f}), \tag{3.15}$$

where $A_{f,j_{m,f}}(x)$ is the membership function (MF) associated with the fuzzy set $A_{f,j_{m,f}}$.

The *association degree* $h_m(\mathbf{x}_n)$ with the class C_{j_m} is calculated as:

$$h_m(\mathbf{x}_n) = w_m(\mathbf{x}_n) \cdot RW_m \tag{3.16}$$

Different definitions have been proposed for the rule weight RW_m [89, 91]. As discussed in [88], the rule weight of each fuzzy rule R_m can improve the performance of FRBCs. For our algorithm, we adopt the fuzzy confidence value, or certainty factor (CF), defined as follows:

$$RW_m = CF_m = \frac{\sum_{\mathbf{x}_n \in C_{j_m}} w_m(\mathbf{x}_n)}{\sum_{n=1}^{N} w_m(\mathbf{x}_n)} \tag{3.17}$$

where N is the number of input-output pairs contained in the training set T.

An FRBC is also characterized by its *reasoning method*, which uses the information from the RB to determine the class label for a specific input pattern. Two different approaches are often adopted in the literature:

1. *The maximum matching*: an input pattern is classified into the class corresponding to the rule with the maximum association degree calculated for the pattern.
2. *The weighed vote*: an input pattern is classified into the class corresponding to the maximum total strength of vote. In particular, for a new pattern $\widehat{\mathbf{x}}$, the total strength of vote for each class is computed as follows:

$$V_{C_l}(\widehat{\mathbf{x}}) = \sum_{R_m \in RB; C_{j_m} = C_l} h_m(\widehat{\mathbf{x}}) \quad (3.18)$$

where $C_l \in C = \{C_1, \ldots, C_L\}$. With this method, each fuzzy rule gives a vote for its consequent class. If no fuzzy rule matches the pattern $\widehat{\mathbf{x}}$, we classify $\widehat{\mathbf{x}}$ as *unknown*.

3.3.2 Fuzzy association rules for classifications

Association rules are rules in the form $Z \rightarrow Y$, where Z and Y are set of items. These rules describe relations among items in a dataset [79]. Association rules have been widely employed in the market basket analysis. Here, items identify products and the rules describe dependencies among the different products bought by customers [2]. Such relations can be used for decisions about marketing activities as promotional pricing or product placements.

In the associative classification context, the single item is defined as the couple $IT_{f,j} = (X_f, v_{f,j})$, where $v_{f,j}$ is one of the discrete values that variable X_f, $f = 1, ..., F$, can assume. A generic classification association rule CAR_m is expressed as:

$$CAR_m : Ant_m \rightarrow C_{j_m} \quad (3.19)$$

where Ant_m is a conjunction of items, and C_{j_m} is the class label selected for the rule among the set $C = \{C_1, \ldots, C_L\}$ of possible classes. For each variable X_f, just one item is typically considered in Ant_m. Antecedent Ant_m can be represented more familiarly as

$$Ant_m : X_1 \textbf{ is } v_{1,j_{m,1}} \ \ldots \ \textbf{AND} \ \ldots \ X_F \textbf{ is } v_{F,j_{m,F}} \quad (3.20)$$

where $v_{f,j_{m,f}}$ is the value used for variable X_f in rule CAR_m.

Most of the association rule analysis techniques are focused on binary or discrete attributes. However, in the framework of pattern classification, input variables can be also continuous. For continuous variables, a discretization process is used to generate a finite set of Q_f atomic values $V_f = v_{f,1}, \ldots, v_{f,Q_f}$ associated with the specific variable X_f. In this context, fuzzy set theory can offer a very suitable tool for approaching the discretization problem, ensuring a high interpretability of the rules, thanks to the use of linguistic terms, and avoiding unnatural boundaries in the partitioning of the attribute domain [6].

In fuzzy associative classification context, given a set of attributes **X** $= \{X_1, \ldots, X_F\}$ and a fuzzy partition P_f defined for each attribute X_f, the single item is defined as the

couple $IT_{f,j} = (X_f, A_{f,j})$, where $A_{f,j}$ is one of the fuzzy values defined in the partition P_f of variable X_f, $f = 1, ..., F$. A generic fuzzy CAR for classification is expressed as:

$$FCAR_m : FAnt_m \rightarrow C_{j_m} \tag{3.21}$$

where C_{j_m} is the class label selected for the rule among the set $C = \{C_1, \ldots, C_L\}$ of possible classes and $FAnt_m$ is a conjunction of items. The antecedent $FAnt_m$ can be represented more familiarly as

$$FAnt_m : X_1 \textbf{ is } A_{1,j_{m,1}} \ \ldots \ \textbf{AND} \ \ldots \ X_F \textbf{ is } A_{F,j_{m,F}} \tag{3.22}$$

where $A_{f,j_{m,f}}$ is the fuzzy value used for variable X_f in rule $FCAR_m$.

In the association rule analysis, support and confidence are the most common measures to determine the strength of an association rule.

Support and confidence can be expressed for a fuzzy rule $FCAR_m$ as follows:

$$fuzzySupp(FAnt_m \rightarrow C_{j_m}) = \frac{\sum_{\mathbf{x}_n \in C_{j_m}} w_m(\mathbf{x}_n)}{N} \tag{3.23}$$

$$fuzzyConf(FAnt_m \rightarrow C_{j_m}) = \frac{\sum_{x_n \in C_{j_m}} w_m(\mathbf{x}_n)}{\sum_{\mathbf{x}_n \in T} w_{Ant_m}(\mathbf{x}_n)} \tag{3.24}$$

where T is the training set, N is the number of objects in T, $w_m(\mathbf{x}_n)$ is the matching degree of rule $FCAR_m$ and $w_{FAnt_m}(\mathbf{x}_n)$ is the matching degree of all the rules which have the antecedent equal to $FAnt_m$.

3.4 AC-FFP: a novel Associative Classification model based on a Fuzzy Frequent Pattern mining algorithm

The use of fuzzy partitions makes the fuzzy CAR mining more complex. Indeed, while in the case of crisp partitions an input value supports a unique item, in the case of fuzzy partitions, an input value can support more than one fuzzy item (in our implementation, which is based on strong fuzzy partitions, the supported items are at most two). Thus, the number of possible fuzzy association rules is higher than the number of possible crisp rules. The approaches proposed so far in the literature for generating fuzzy association rules have limited the complexity by considering only the most frequent fuzzy item for each attribute [37, 109]. Obviously, this solution reduces the number of association rules, but also the amount of information described by these rules. In this context, we aim to exploit the advantages of fuzzy set theory in terms of modeling capability, without dramatically reducing the complexity and therefore the information.

In this section, we propose a new efficient fuzzy association rule-based classification scheme, which mines fuzzy CARs by using a fuzzy version of the well-known FP-Growth algorithm [80], called *AC-FFP*. Even though some fuzzy versions of FP-Growth have been already proposed in the literature [37, 109], our method represents the first attempt

of using such algorithm for deriving fuzzy CARs. Indeed, the works in [37, 109] just propose a fuzzy version of the FP-Growth algorithm aimed at mining fuzzy association rules for descriptive modeling rather than for classification. Moreover, we aim to propose an approach that is easy to implement, is computationally light and guarantees to achieve accuracy values comparable with other state-of-the-art approaches. Similar to MRAC and MRAC+, the ratio behind this approach is to build an accurate and fast fuzzy associative classifier that is able to handle a huge amount of data. For this reason, unlike some of the recent contributions discussed above, our fuzzy associative classification scheme does not use any additional optimization algorithm (such as the evolutionary algorithm in [6, 63, 147] or the simulated annealing employed in [125]), thus maintaining the time required to generate the fuzzy association rules acceptable. Furthermore, the FP-Growth algorithm results more scalable than the Apriori algorithm used in [123, 151, 163] with respect to the number of instances and attributes.

We have proposed a set of appropriate novel strategies, which allow us to efficiently generate accurate fuzzy associative classifiers. In particular, the main novelties introduced in the proposed fuzzy association rule-based classification scheme are:

- A novel approach to define strong fuzzy partitions from crisp partitions obtained by applying the classical Fayyad and Irani discretization algorithm [61].
- The extension of the FP-Growth algorithm to the fuzzy context for mining a set of fuzzy CARs. In particular, we adopt proper definitions of fuzzy support and confidence. Further, we consider, for each attribute, all the frequent fuzzy sets rather than only the most frequent when generating the fuzzy CARs. Finally, we just adopt the FP-Growth algorithm. Most of the proposed fuzzy associative classifiers, such as the ones described in [6, 34, 63, 123, 149], are based on the Apriori algorithm. This algorithm, as discussed in Section 3.3, is characterized by a number of weaknesses, especially when dealing with large and high-dimensional datasets.
- Three purposely adapted types of fuzzy CAR pruning. The first type considers fuzzy support and confidence with respect to two thresholds, namely *minSupp* and *minConf*. These thresholds are adapted to the number of conditions and number of instances of each class, respectively, so as to take into account the effect of the specific implementation of the conjunction operator and the imbalance of datasets. The second type removes redundant rules based on fuzzy support and confidence, and rule length. The third type exploits the training set coverage: only the fuzzy rules, which are activated by at least one data object in the training set, are retained.
- An adjustment of the weighted vote reasoning method for classifying unlabeled patterns: the vote of each single rule is modified accordingly to its rule length. This modification balances the relevance of more general and more specific rules, thus improving the overall classification accuracy.

We compare the results achieved by the proposed approach on seventeen datasets with the ones obtained by Li2001CMAR [108], an associative classifier based on the FP-Growth algorithm. By using non-parametric statistical tests, we show that our ap-

proach outperforms CMAR in terms of accuracy. Further, we compare the proposed fuzzy associative classifier with two recent state-of-the-art approaches, namely FARC-HD [6] and D-MOFARC [63], for mining fuzzy CARs. We show that our approach is statistically equivalent to the comparison approaches. On the other hand, we have to highlight that FARC-HD and D-MOFARC employ a fuzzy adaptation of the Apriori algorithm for mining the fuzzy rules and an evolutionary post-processing for pruning these rules and optimizing the fuzzy partitions. Thus, our fuzzy associative classification scheme, based on the fuzzy FP-Growth, results to be more scalable, especially when dealing with large and high dimensional datasets.

The rest of the section is organized as follows. Section 3.4.1 describes each phase of the proposed approach and includes the details of the fuzzy FP-Growth algorithm. Section 3.4.2 presents the experimental setup and discusses the results that are obtained on seventeen real-world datasets.

3.4.1 The Proposed Approach

In this section, we present our Associative Classifier based on a Fuzzy Frequent Pattern (AC-FFP) mining algorithm. AC-FFP consists of the following three phases:

1. *Discretization*: a fuzzy partition is defined on each linguistic variable by using the multi-interval discretization approach based on entropy proposed by Fayyad and Irani in [61];
2. *Fuzzy CAR Mining*: a fuzzy frequent pattern mining algorithm, which is an extension of the well known FP-Growth, is exploited to extract frequent fuzzy classification rules with confidence higher than a pre-fixed threshold;
3. *Pruning*: rule pruning based on redundancy and training set coverage is applied to generate the final RB.

At the end of the three phases, we obtain an FRBC, which can be used for the classification task of unlabeled patterns.

In the following, we introduce in detail all the mentioned phases.

Discretization

The discretization of continous features is a critical aspect in the generation of association rule classifiers. In the last years, several different heuristic methods have been proposed [39, 51, 61, 100]. We use the method proposed by Fayyad and Irani in [61]. This supervised method exploits the class information entropy of candidate partitions to select the bin boundaries for discretization.

Let $T_{f,0} = [x_{1,f}, ..., x_{N,f}]^T$ the projection of the training set T along variable X_f and $b_{f,r}$ a bin boundary for the same variable. Let $T_{f,1}$ and $T_{f,2}$ be the subsets of points of the set $T_{f,0}$ which lie in the two bins identified by $b_{f,r}$. The class information entropy of the discretization induced by $b_{f,r}$, denoted as $E(X_f, b_{f,r}; T_{f,0})$ is given by

$$E(X_f, b_{f,r}; T_{f,0}) = \frac{|T_{f,1}|}{|T_{f,0}|} \cdot Ent(T_{f,1}) + \frac{|T_{f,2}|}{|T_{f,0}|} \cdot Ent(T_{f,2}) \tag{3.25}$$

where $|\cdot|$ denotes the cardinality and $Ent(\cdot)$ is the entropy calculated for a set of points [61]. The boundary $b_{f,min}$, which minimizes the class information entropy over all possible partition boundaries $b_{f,r}$ of $T_{f,0}$ is selected as a binary discretization boundary. The method is then applied recursively to both the partitions induced by $b_{f,min}$ until the following stopping criterion based on the Minimal Description Length Principle is achieved. Recursive partitioning stops iff

$$Gain(X_f, b_{f,min}; T_{f,0}) < \frac{\log_2(|T_{f,0}| - 1)}{|T_{f,0}|} + \frac{\Delta(X_f, b_{f,min}; T_{f,0})}{|T_{f,0}|} \tag{3.26}$$

where

$$Gain(X_f, b_{f,min}; T_{f,0}) = Ent(T_{f,0}) - E(X_f, b_{f,min}; T_{f,0}), \tag{3.27}$$

$$\Delta(X_f, b_{f,min}; T_{f,0}) = \log_2(3^{k_0} - 2) - [k_0 \cdot Ent(T_{f,0}) - k_1 \cdot Ent(T_{f,1}) - k_2 \cdot Ent(T_{f,2})] \tag{3.28}$$

and k_i is the number of class labels represented in the set $T_{f,i}$.

The method outputs, for each variable, a set of bin boundaries. Let $U_f = [x_{f,l}, x_{f,u}]$ be the universe of variable X_f. Let $\{b_{f,1}, \ldots, b_{f,Q_f}\}$, with $\forall r \in [1, \ldots, Q_f - 1], b_{f,r} < b_{f,r+1}$, be the set of bin boundaries, where $b_{f,1} = x_{f,l}$ and $b_{f,Q_f} = x_{f,u}$. Then, the method identifies the set $\{[b_{f,1}, b_{f,2}], \ldots, [b_{f,Q_f-1}, b_{f,Q_f}]\}$ of contiguous intervals, which partition the universe of variable X_f.

To transform the crisp partition into a strong fuzzy partition, we adopt the following procedure. For each bin $[b_{f,r}, b_{f,r+1}]$, with $r \in [1, \ldots, Q_f - 1]$, we first compute the middle point $m_{f,r} = \frac{b_{f,r}+b_{f,r+1}}{2}$ and then generate three triangular fuzzy sets $A_{f,2r-1}$, $A_{f,2r}$ and $A_{f,2r+1}$ defined as $(m_{f,r-1}, b_{f,r}, m_{f,r})$, $(b_{f,r}, m_{f,r}, b_{f,r+1})$ and $(m_{f,r}, b_{f,r+1}, m_{f,r+1})$, respectively. We recall that a triangular fuzzy set is defined by three points (a, b, c), where b represents the core and a and c correspond to the lower and upper bounds of the support, respectively. The two fuzzy sets $A_{f,1}$ and $A_{f,2Q_f-1}$ at the lower and upper bounds of the universe of X_f are defined as $A_{f,1} = (-\infty, b_{f,1}, m_{f,1})$ and $A_{f,2Q_f-1} = (b_{f,Q_f-1}, m_{Q_f-1}, +\infty)$, respectively. The set $P_f = \{A_{f,1}, \ldots, A_{f,T_f}\}$, where $T_f = 2Q_f - 1$ is the number of fuzzy sets for each feature, defines the fuzzy partition of feature X_f. If no bin boundary has been found by the algorithm for feature X_f, then no fuzzy value is generated for this feature and the feature is discarded. Figure 3.15 shows an example of strong fuzzy partition obtained by the fuzzification of the output of the Fayyad and Irani's discretizer.

As shown in Fig. 3.15 and discussed in the text, the cores of the triangular fuzzy sets are positioned in correspondence to both the middle points and the bin boundaries. We performed different experiments for determining the best number of fuzzy sets and also the best positioning of these fuzzy sets. For instance, we generated strong fuzzy partitions by using only the middle points or only the bin boundaries. We verified that the best results in terms of accuracy are obtained by using fuzzy sets positioned on both middle points and bin boundaries. On the other hand, the fuzzy sets positioned on the middle

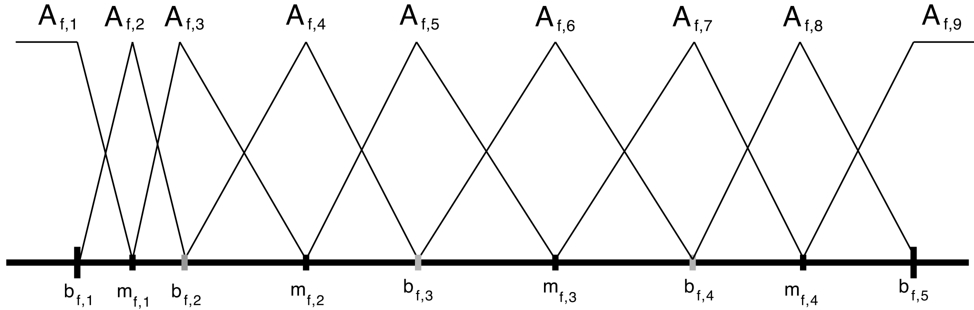


Figure 3.15: An example of strong fuzzy partition obtained by the fuzzification of the output of the Fayyad and Irani's discretizer.

points allow modeling accurately the instances belonging to the bin and consequently the class connected to the bin. Further, the fuzzy sets on the bin boundaries permit to finely discriminate instances belonging to two different bins and possibly different classes.

Fuzzy CAR Mining

To mine the fuzzy CARs from the dataset, we introduce a novel fuzzy frequent pattern (FFP) mining algorithm. This algorithm is based on the well known FP-Growth proposed by Han et al. in [80] for efficiently mining frequent patterns without generating candidate itemsets. The algorithm consists of two phases. The first phase creates an FP-tree from the dataset and the second phase extracts frequent patterns from the FP-tree. The creation of the FP-tree is performed in three steps. First, the dataset is scanned to find the frequent items. Then, these items are sorted in descending frequency. Finally, the dataset is scanned again to construct the FP-tree according to the sorted order of frequent items.

In the second phase, all frequent itemsets are mined from the FP-tree. For each item, a conditional FP-tree is generated and from this tree the frequent itemsets, including the processed item, are recursively mined.

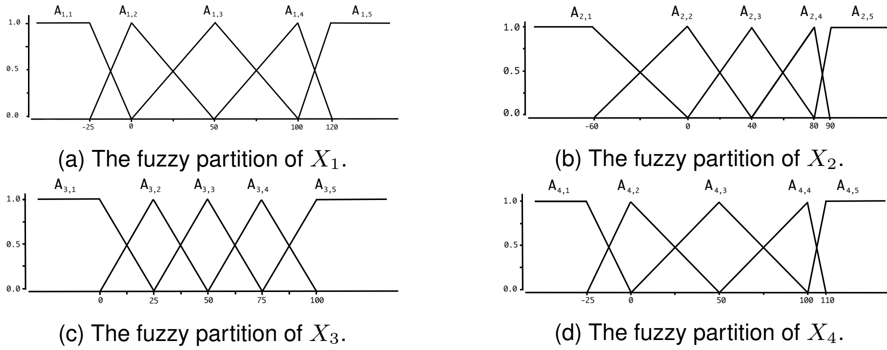
Some papers have already proposed to integrate the fuzzy theory with the FP-Growth algorithm. In [37] the authors choose only the most frequent linguistic value for each variable to build the FP-tree. For example, if the fth partition contains T_f fuzzy sets, only one of these fuzzy sets is used to mine rules. Thus, only a limited subset of rules is generated and therefore useful information resulting from other fuzzy items might be removed. A similar approach is presented in [139]. Unlike these approaches, in AC-FFP we try to preserve information as much as possible.

AC-FFP performs four scans of the dataset. The first two scans determine the fuzzy frequent values and build the FP-tree, respectively. The FP-tree is therefore used to mine fuzzy frequent patterns and then fuzzy CARs. The third and fourth scans are needed to compute fuzzy support and confidence, and the training set coverage, respectively, in

the pruning phase. In the following, we will describe in detail the operations performed in the four scans with the help of an example of application. In the example, we adopt the training set shown in Table 3.9. Further, we assume that the discretization and the subsequent fuzzification process have partitioned the input variables as in Figure 3.16.

Table 3.9: A simple dataset characterized by four input features.

ID	X_1	X_2	X_3	X_4	**Class**
1	20	20	0	10	C_1
2	25	-60	10	80	C_3
3	-25	40	100	40	C_1
4	75	60	35	110	C_2
5	20	80	100	75	C_2
6	30	90	75	10	C_3
7	120	50	75	-25	C_1



(a) The fuzzy partition of X_1.

(b) The fuzzy partition of X_2.

(c) The fuzzy partition of X_3.

(d) The fuzzy partition of X_4.

Figure 3.16: The fuzzy partitions of each variable in the example.

In the first scan, AC-FFP calculates the fuzzy support of each fuzzy value $A_{f,j}$. The fuzzy support is computed as:

$$fuzzySupp(A_{f,j}) = \frac{\sum_{n=1}^{N} A_{f,j}(x_{f,n})}{N} \tag{3.29}$$

Only the fuzzy values, called *frequent fuzzy values*, whose support is larger than the support threshold $minSup$ (0.2 in the example) are retained and organized in a list, called f_{list}, in support descending order. The other fuzzy values are pruned and therefore not considered in the fuzz CAR mining. In Table 3.10, we show the fuzzy supports calculated for each fuzzy set considered in the example. From the analysis of Table 3.10, the following f_{list} is generated:

f_{list} = {$A_{2,3}$, $A_{1,3}$, $A_{1,2}$, $A_{4,3}$, $A_{3,4}$, $A_{3,5}$, $A_{4,2}$, $A_{2,4}$, $A_{3,1}$ }.

In the second scan, AC-FFP builds the FP-tree in order to mine all the fuzzy CARs. The generation of the FP-tree is performed as in FP-Growth: the only difference is that

Table 3.10: The fuzzy supports of each fuzzy set in the example.

Fuzzy Value	Fuzzy Support	Fuzzy Value	Fuzzy Support
$A_{1,1}$	0.14	$A_{3,1}$	0.23
$A_{1,2}$	0.30	$A_{3,2}$	0.14
$A_{1,3}$	0.34	$A_{3,3}$	0.06
$A_{1,4}$	0.07	$A_{3,4}$	0.29
$A_{1,5}$	0.14	$A_{3,5}$	0.29
$A_{2,1}$	0.14	$A_{4,1}$	0.14
$A_{2,2}$	0.07	$A_{4,2}$	0.26
$A_{2,3}$	0.38	$A_{4,3}$	0.30
$A_{2,4}$	0.25	$A_{4,4}$	0.16
$A_{2,5}$	0.14	$A_{4,5}$	0.14

here the items correspond to fuzzy values. Actually, if we consider the example fuzzy partition in Figure 3.15, we can observe that each value on the universe belongs to two different fuzzy values with different membership grades. Thus, two fuzzy values should be associated with each value. However, if we associate two fuzzy values for each feature value, each object would generate 2^F patterns.

To limit the number of possible patterns, we assign each continuous value to the fuzzy set with the highest membership value (in case of tie, we randomly select one of the two fuzzy sets). Each object $\mathbf{x}_n$ is therefore transformed into a fuzzy object $\widetilde{\mathbf{x}}_n = \{A_{1_n,j_{1_n}}, \ldots, A_{Z_n,j_{Z_n}}\}$, where $A_{i_n,j_{i_n}}$, $i_n \in [1,\ldots,F]$, $j_{i_n} \in [1,\ldots T_{i_n}]$, indicates the frequent fuzzy value selected for feature i_n. The fuzzy values in $\widetilde{\mathbf{x}}_n$ are sorted in the same order as in the f_{list}, as required by the FP-Growth algorithm. Obviously, the number of features, which describe the fuzzy object, can be lower than F. Table 3.11 shows for each pattern of the example dataset, the fuzzy values associated with the highest membership degree and the corresponding fuzzy objects for each pattern in the example training set.

Table 3.11: The fuzzy values associated with the highest membership degree and the corresponding fuzzy objects for each pattern in the example dataset.

ID	X_1	X_2	X_3	X_4	$\widetilde{\mathbf{x}}_\mathbf{n}$	Class
1	$A_{1,2}$	$A_{2,3}$	$A_{3,1}$	$A_{4,2}$	$(A_{2,3}\ ,\ A_{1,2}\ ,\ A_{4,2}\ ,\ A_{3,1})$	C_1
2	$A_{1,3}$	$A_{2,1}$	$A_{3,1}$	$A_{4,4}$	$(A_{1,3}\ ,\ A_{3,1})$	C_3
3	$A_{1,1}$	$A_{2,3}$	$A_{3,5}$	$A_{4,3}$	$(A_{2,3}\ ,\ A_{4,3}\ ,\ A_{3,5})$	C_1
4	$A_{1,4}$	$A_{2,4}$	$A_{3,2}$	$A_{4,5}$	$(A_{2,4})$	C_2
5	$A_{1,2}$	$A_{2,4}$	$A_{3,5}$	$A_{4,3}$	$(A_{1,2}\ ,\ A_{4,3}\ ,\ A_{3,5}\ ,\ A_{2,4})$	C_2
6	$A_{1,3}$	$A_{2,5}$	$A_{3,4}$	$A_{4,2}$	$(A_{1,3}\ ,\ A_{3,4}\ ,\ A_{4,2})$	C_3
7	$A_{1,5}$	$A_{2,3}$	$A_{3,4}$	$A_{4,1}$	$(A_{2,3}\ ,\ A_{3,4})$	C_1

The fuzzy objects are used to build the FP-tree. Each branch from the root to a leaf node describes a fuzzy rule. When a fuzzy object of the training set is added to the FP-tree, the fuzzy values are considered as labels: if a node already exists, the corresponding counter is simply incremented by 1. Figure 3.17 shows the FP-tree generated after the fuzzy CAR mining process on the example dataset.

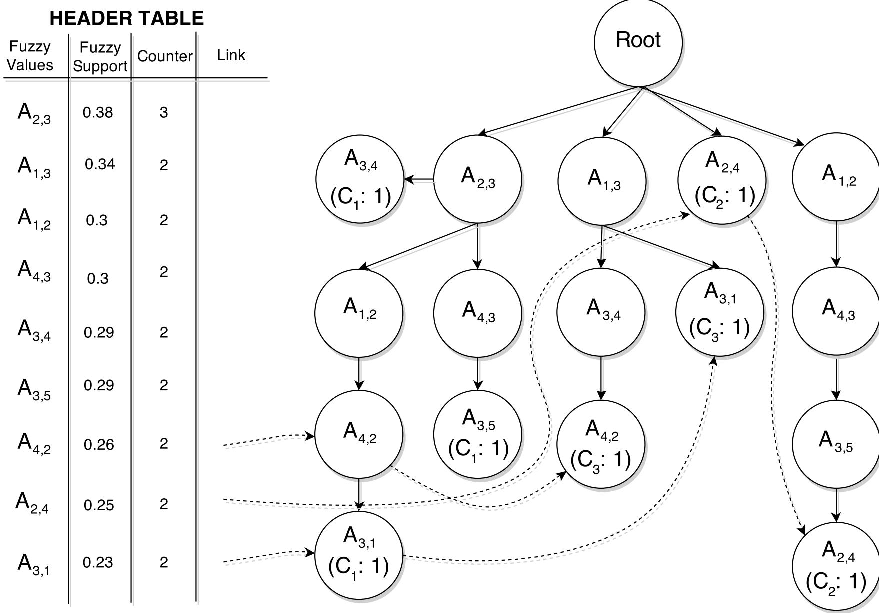


Figure 3.17: The FP-tree generated by using the example dataset.

As in the CMAR algorithm, which is an associative classification model based on the classical version of FP-Growth [108], the rules are extracted from the FP-tree by using $minSupp$ and $minConf$. In particular, the association rules, which are not characterized by a support and a confidence higher than $minSupp$ and $minConf$, respectively, are first generated and therefore eliminated. We recall that support and confidence are here computed by only considering the frequency of the fuzzy values. Further, similar to the pruning process discussed in [108] for CMAR, we test whether the antecedent of each rule is positively correlated with the consequent class by performing the χ^2 test. Only the rules with a χ^2 value higher than $min\chi^2$ are maintained. Figure 3.18 shows the pseudo code of the fuzzy CAR mining process.

Since we consider only the fuzzy objects for generating the FP-tree, only a high quality subset of rules is stored in the $FCAR_{list}$. Indeed, each rule $FCAR_m$ mined from the FP-tree represents the rule with the highest matching degree for the specific object $\mathbf{x}_n \in T$. Other rules, which could be mined from $\mathbf{x}_n$, would have had a lower matching degree and probably would have been pruned. At the end of the second scan, list $FCAR_{list}$ still contains a large amount of fuzzy CARs that are pruned in the subsequent phase.

Pruning

Rule pruning aims to discard slightly relevant rules so as to speed up the classification process. Pruning has to be applied carefully since an excessive elimination of rules may delete useful knowledge. Several approaches to rule pruning have been proposed in

Data: $f_{list}, minSupp, minConf, min\chi^2$

Result: a list $FCAR_{list}$ of FCARs

$FP_{tree} \leftarrow$ clear;

$FCAR_{list} \leftarrow$ clear;

forall *data objects* $(\mathbf{x}_n, y_n)$ *in the training set* **do**

    forall *input feature values* $x_{n,f}$ **do**

        $\widetilde{x}_{n,f} \leftarrow$ fuzzy value with maximum membership grade for $x_{n,f}$;

        if *($\widetilde{x}_{n,f}$ is in the f_{list}, that is, $\widetilde{x}_{n,f}$ is frequent)* **then**

            $\widetilde{x}_n \leftarrow \widetilde{x}_{n,f}$;

        end

    end

    sort $\widetilde{\mathbf{x}}_n$ according to f_{list};

    insert $(\widetilde{\mathbf{x}}_n, y_n)$ into FP_{tree};

end

$FCAR_{list} \leftarrow$ mineAllRules($FP_{tree}, minSupp, minConf, min\chi^2$);

Figure 3.18: Pseudo-code of the fuzzy CAR mining process based on FP-Growth.

the last years, such as lazy pruning [16], database coverage [113] and pessimistic error estimation [197].

We perform three different types of pruning. In the first type, a rule $FCAR_m$ is pruned if its fuzzy support and confidence are not higher than *minFuzzySupp* and *minFuzzyConf*, respectively. These thresholds correspond to *minSupp* and *minConf* adapted to the number of conditions and number of instances of each class, respectively, so as to take into account both the effect of the t-norm used as conjunction operator and the imbalance of datasets.

Indeed, since we use the product as t-norm for implementing the conjunction operator, rules with a higher number of conditions in the antecedent will be characterized by a lower support value than rules with a lower number of conditions. Actually, this result is mainly due to the effect of the t-norm rather than to the activation of each condition. Indeed, each condition could be activated with a high matching degree, but for the behavior of the t-norm the matching degree of the rule, when the number of conditions is high, would result to be quite low. With the aim of reducing this effect and therefore avoiding to penalize more specific rules, we adapt the threshold *minFuzzySupp* on the fuzzy support to the rule length (RL), that is, the number of conditions in the antecedent of the rules, as follows:

$$minFuzzySupp_g = minSupp \cdot 0.5^{g-1} \tag{3.30}$$

where $minSupp$ is the minimum support determined by the expert and $g \in [1..F]$ is the rule length.

For example, for a rule with one condition in the antecedent, we have $minFuzzySupp_1 = minSupp$. For a rule with two conditions in the antecedent, we have $minFuzzySupp_2 = minSupp \cdot 0.5$, and so on.

To take also the imbalanced datasets into consideration, the confidence threshold is adapted by considering the imbalance ratio between each class and the majority class as follows:

$$minFuzzyConf_{C_j} = minConf \cdot \frac{N_{C_j}}{N_{MajorityClass}} \tag{3.31}$$

where $N_{MajorityClass}$ is the number of occurrences of the majority class label in the data set, N_{C_j} is the number of occurrences of the consequent class C_j in the training set and $minConf$ is the minimum confidence fixed by the expert. Formula 3.31 allows decreasing the $minConf$ threshold proportionally to the imbalance ratio between the class of the rule and the majority class. Thus, rules, which have a minority class in the consequent, are not pruned only because the number of instances of that class is very low in the training set.

Figure 3.19 shows the pseudo-code of the first type of pruning, which involves the third scan of the dataset.

With the adjustments performed by formulas (3.30) and (3.31), we are able to mine a higher number of fuzzy rules than the other approaches described in [37] and [139], without losing the advantages of the FP-Growth method, even if a third scan in the dataset is necessary.

In the second type of pruning, redundant rules are removed. First, the rules are sorted according to the fuzzy support, confidence and RL. In particular, rule $FCAR_s$ has higher rank than rule $FCAR_m$, if and only if:

1. $fuzzyConf(FCAR_s) > fuzzyConf(FCAR_m)$
2. $fuzzyConf(FCAR_s) = fuzzyConf(FCAR_m)$ **AND** $fuzzySupp(FCAR_s) > fuzzySupp(FCAR_m)$;
3. $fuzzyConf(FCAR_s) = fuzzyConf(FCAR_m)$ **AND** $fuzzySupp(FCAR_s) = fuzzySupp(FCAR_m)$ **AND** $RL(FCAR_s) < RL(FCAR_m)$.

A fuzzy rule $FCAR_m$ is pruned if and only if there exists a rule $FCAR_s$ with higher rank and more general than $FCAR_m$. A rule $FCAR_s : FAnt_s \rightarrow C_{j_s}$ is more *general* than a rule $FCAR_m : FAnt_m \rightarrow C_{j_m}$, if and only if, $FAnt_m \subseteq FAnt_s$. Our experimental results show that this second step can reduce significantly the number of fuzzy CARs in the $FCAR_{list}$.

In the third type of pruning, the *training set coverage* is exploited: only the fuzzy rules that are activated by at least one data object in the training set are retained. Each data object in the training set is associated with a counter initialized to 0. For each object, a scan over the sorted $FCAR_{list}$ is performed to find all the rules that match the object: we consider only those rules $FCAR_m$ with matching degree higher than the *fuzzy matching degree threshold* $\overline{w}_m = 0.5^{g_m - 1}$, where g_m is the rule length of $FCAR_m$. This threshold

Data: $FCAR_{list}, minSupp, minConf$

Result: Pruned $FCAR_{list}$

forall *data objects* $(\mathbf{x}_n, y_n)$ *in the training set* **do**

  forall *fuzzy rules* $FCAR_m$ *in* $FCAR_{list}$ **do**

    $w_m(\mathbf{x}_n) \leftarrow$ calculate matching degree of $\mathbf{x}_n$ with $FCAR_m$;

  end

end

forall *possible rule lengths g in [1..F]* **do**

  compute $minFuzzySupp_g$ by using formula(17);

end

forall *classes* C_j *in* $[C_1, ..., C_L\ ]$ **do**

  compute $minFuzzyConf_{C_j}$ by using formula(18);

end

forall *fuzzy rules* $FCAR_m$ *in* $FCAR_{list}$ **do**

  $g_m \leftarrow$ rule length of $FCAR_m$

  $C_{j_m} \leftarrow$ class of $FCAR_m$;

  if $fuzzySupp(FCAR_m) < minFuzzySupp_{g_m}$ **OR**

  $fuzzyConf(FCAR_m) < minFuzzyConf_{C_{jm}}$ **then**

    $FCAR_{list} \leftarrow$ remove $FCAR_m$ from $FCAR_{list}$;

  end

end

Figure 3.19: Pseudo-code of the first type of pruning.

allows us to take into account only the most significant rules for a specific data object, without penalizing rules with high rule length. If $FCAR_m$ classifies correctly at least one data object, then it is inserted into the RB. Further, the counters associated with the objects, which activate $FCAR_m$, are incremented by 1. Whenever the counter of an object becomes larger than the *coverage threshold* δ, the data object is removed from the training set and no longer considered for subsequent rules. Since rules are sorted in descending ranks, it is very likely that these subsequent rules would have a very limited relevance for the object. The procedure ends when no more objects are in the training set or all the rules have been analyzed. Figure 3.20 shows the pseudo-code of the third type of pruning.

We would like to point out that the overall rule base is obtained by performing only 4 scans of the overall training set.

```
Data: sorted pruned FCAR_list, coverage threshold δ
Result: final RB
RB ← clear();
forall data objects (x_n, y_n) in the training set do
    count ← 0;
    forall rules FCAR_m in FCAR_list do
        w_m(x_n) ← calculate the matching degree;
        if w_m(x_n) >= w̄_m then
            count ← count + 1;
            if FCAR_m correctly classifies x_n AND
            FCAR_m is not in RB then
                add FCAR_m to RB;
            end
        end
        if count > δ then
            break;
        end
    end
end
```

Figure 3.20: Pseudo-code of the third type of pruning.

Classification

The set of rules survived after the pruning are used to classify unlabeled patterns. We adopt the weighted vote [40] as *reasoning method*: an input pattern is classified into the class corresponding to the the maximum total strength of vote, calculated by using formula (3.18). Given an input pattern $\widehat{\mathbf{x}} = [\hat{x}_1 \ldots, \hat{x}_F]$, each fuzzy rule in the RB gives a vote for its consequent class. If $\widehat{\mathbf{x}}$ activates no rule, then $\widehat{\mathbf{x}}$ is classified as unknown.

Since we use the product t-norm as conjunction operator, rules with a higher number of conditions in the antecedent have generally a lower matching degree than rules with a lower number of conditions in the antecedent. Hence, more general rules are more influential than specific rules in the prediction phase. To re-balance the influence, we normalize formula (3.18) as follows:

$$V_{C_k}(\widehat{\mathbf{x}}) = \sum_{FCAR_m \in RB; C_{j_m} = C_l} w_m(\widehat{\mathbf{x}}) \cdot 2^{g_m} \cdot CF_m \tag{3.32}$$

where $w_m(\widehat{\mathbf{x}})$ is the matching degree of $FCAR_m$ for the input $\widehat{\mathbf{x}}$, F is the number of features in the data set, g_m is the RL of $FCAR_m$ and CF_m is the certainty factor. For example, let us assume that three rules have, respectively, one condition, two conditions and three conditions. Let us suppose that each object is described by 3 features and each condition is activated by the unlabeled pattern $\widehat{\mathbf{x}}$ with membership degree equal to 0.5. The matching degrees for the three rules would be 0.5, 0.25 and 0.125, respectively. After re-balancing with formula (3.32), all the votes are equal to 1.

This normalization allows considering also the vote of rules with a high number of conditions. Indeed, the vote of these rules is strongly penalized by the number of conditions, even if all these conditions are activated with a high membership degree. In our experiments, we verified that this approach is more effective than, for instance, maximum matching.

3.4.2 Experimental Study

We tested our method on seventeen classification datasets extracted from the KEEL repository[4]. As shown in Table 3.12, the datasets are characterized by different numbers of input variables (from 4 to 16), input/output instances (from 106 to 19020) and classes (from 2 to 11). For the datasets CLE and WIS, we removed the instances with missing values. The number of instances in the table refers to the datasets after the removing process.

Table 3.12: Datasets used in the experiments (sorted for increasing numbers of input variables).

Dataset	# Instances	# Variables	# Classes
Iris (IRI)	150	4	3
Phoneme (PHO)	5404	5	2
Newthyroid (NEW)	215	5	3
Monk-2 (MON)	432	6	2
Appendicitis (APP)	106	7	2
Ecoli (ECO)	336	7	8
Pima (PIM)	768	8	2
Yeast (YEA)	1484	8	10
Glass(GLA)	214	9	6
Wisconsin (WIS)	683	9	2
Page-Blocks (PAG)	5472	10	5
Magic (MAG)	19020	10	2
Heart (HEA)	270	13	2
Cleveland (CLE)	297	13	5
Wine (WIN)	178	13	3
Vowel (VOW)	990	13	11
Pen-Based (PEN)	10992	16	10

[4] Available at `http://sci2s.ugr.es/keel/datasets.php`

We compare the results obtained by AC-FFP with the ones achieved by three different classification models, namely CMAR [108], FARC-HD [6] and D-MOFARC [63]. We chose these three algorithms because CMAR exploits as AC-FFP the FP-Growth algorithm for generating the association rules, and FARC-HD and D-MOFARC are, to the best of our knowledge, two of the most recent and effective fuzzy rule-based associative classifiers proposed in the literature.

Similar to our approach, CMAR first adopts the multi-interval discretization method presented in [61] to split the input domains into bins. Then, it builds a class distribution-associated frequent pattern tree to efficiently mine CARs. Finally, CARs are pruned based on the analysis of the: i) confidence, ii) correlation, iii) rule redundancy and iv) database coverage. The classification is performed based on a weighted χ^2 analysis enforced on multiple association rules. We implemented a JAVA version of CMAR following the description provided in [108].

FARC-HD and D-MOFARC have been described in Section 3.3. In [6], the authors have shown that FARC-HD is very efficient since it outperforms a large number of classical classification algorithms, both based and not based on fuzzy rules and/or on CARs [6]. We recall that D-MOFARC is a recent extension of FARC-HD. In the experiments, we have used the implementations of FARC-HD available in the KEEL package [7]. As regards D-MOFARC, we extracted the results from the paper in which the method has been discussed.

Table 3.13 shows the parameters used for each algorithm in the experiments. The parameters have been chosen according to the guidelines provided by the authors in the papers in which each algorithm has been introduced. For FARC-HD and D-MOFARC, the descriptions of the specific parameters can be found in [6] and [63]. Further, for each dataset and for each algorithm, we performed a ten-fold cross-validation by using the same folds for all the datasets.

Table 3.13: Values of the parameters for each algorithm used in the experiments.

Method	**Parameters**
CMAR	$MinSupp = 0.01, MinConf = 0.5, \delta = 4, min\chi^2 = 20\%$
FARC-HD and D-MOFARC	$MinSupp = 0.05, MaxConf = 0.80, Depth_{max} = 3, k_t = 2,$ $Pop = 50, Iterations = 15.000, BITSGENE = 30, \delta = 2$
AC-FFP	$MinSupp = 0.01, MinConf = 0.5, \delta = 4, min\chi^2 = 20\%$

Table 3.14 shows, for each dataset and for each algorithm, the average values of the accuracy, both on the training (Acc_{Tr}) and test sets (Acc_{Ts}), obtained by the associative classifiers generated by the three algorithms. For each dataset, the values of the highest accuracies are shown in bold. In Table 3.14 the results achieved by D-MOFARC for 4 datasets are missing because these datasets have not been considered in the experiments carried out in [63]. On the other hand, as stated in [49], even though we consider just 13 datasets, the significance of the Wilcoxon signed-rank test is ensured.

From Table 3.14, we can observe that, in most of the datasets, the AC-FFP algorithm generates classifiers more accurate than the ones generated by CMAR. In particular, in

12 out of 17 datasets, AC-FFP achieves the highest accuracies on the test set. As regards FARC-HD and D-MOFARC, we observe that AC-FFP, FARC-HD and D-MOFARC achieve similar average accuracies on the test set. Further, FARC-HD and D-MOFARC suffer from overtraining more than the other comparison approaches.

Table 3.14: Average results obtained by CMAR, FARC-HD, D-MOFARC and AC-FFP.

	CMAR		**FARC-HD**		**D-MOFARC**		**AC-FFP**	
Dataset	Acc_{Tr}	Acc_{Ts}	Acc_{Tr}	Acc_{Ts}	Acc_{Tr}	Acc_{Ts}	Acc_{Tr}	Acc_{Ts}
IRI	96.00	93.33	**98.59**	96.00	98.1	96.0	96.15	**98.00**
PHO	79.10	78.70	83.50	82.14	**84.8**	**83.5**	81.54	81.10
NEW	96.38	93.10	98.98	93.95	**99.8**	95.5	97.73	**95.87**
MON	77.78	77.56	**99.92**	**99.77**	-	-	97.22	97.27
APP	90.87	**86.00**	**93.82**	84.18	-	-	91.51	85.09
ECO	83.30	76.83	92.33	82.19	**94.0**	82.7	89.35	**83.39**
PIM	78.69	74.87	**82.90**	**75.66**	82.3	75.5	79.25	74.87
YEA	56.55	54.32	**63.81**	**58.50**	-	-	57.99	55.60
GLA	80.42	69.44	81.10	70.24	**95.2**	70.6	83.59	**74.18**
WIS	97.74	**96.80**	98.70	96.52	98.6	96.8	**98.85**	96.06
PAG	93.79	93.68	95.62	95.01	**97.8**	**97.0**	94.15	93.88
MAG	79.39	78.94	**85.36**	84.51	86.3	**85.4**	73.59	73.40
HEA	90.08	84.07	93.91	84.44	**100.0**	**90.0**	94.77	81.85
CLE	54.40	53.88	88.18	55.24	**90.9**	52.9	80.43	**56.91**
WIN	99.94	96.05	99.94	94.35	100.0	95.8	**100**	**97.12**
VOW	74.14	61.41	80.48	71.82	-	-	**98.95**	**91.52**
PEN	78.60	77.78	97.04	**96.04**	**97.4**	96.2	85.66	83.48

In order to verify if there exist statistical differences among the values of accuracy on the test set associated with the classifiers generated by the different algorithms, we perform a statistical analysis. As suggested in [49], we apply non-parametric statistical tests combining all the datasets: for each approach we generate a distribution consisting of the mean values of accuracy calculated on the test set. We compare both CMAR, FARC-HD and D-MOFARC with AC-FFP by using a pairwise comparison, namely the Wilcoxon signed-rank test [196], which detects significant differences between two distributions. In all the tests, we use $\alpha = 0.05$ as *level of significance*.

Table 3.15 shows the results of the application of the Wilcoxon signed-rank test between AC-FFP and CMAR, between AC-FFP and FARC-HD, and between AC-FFP and D-MOFARC. As regards the first comparison, since the *p-value* is lower than the significance level α, the null hypothesis of equivalence can be rejected. In conclusion, we can state that AC-FFP statistically outperforms CMAR in terms of classification accuracy on the test set.

As regards the Wilcoxon signed-rank test between AC-FFP and the remaining two approaches, since the *p-value* is higher than the significance level α, the null hypothesis of equivalence is not rejected. Hence, AC-FFP results to be statistically equivalent, in terms of classification rate computed on the test set, to both FARC-HD and D-MOFARC. In FARC-HD and D-MOFARC, however, the steps, which generate the initial set of candidate rules, are based on a fuzzy version of the Apriori algorithm. This algorithm suffers

Table 3.15: Results of the Wilcoxon signed-rank test with a significance level $\alpha = 0.05$.

Comparision	R^+	R^-	P-*value*	*Hypothesis*
AC-FFP vs CMAR	113	23	0.01825	**Rejected**
AC-FFP vs FARC-HD	76	77	1	**Not Rejected**
AC-FFP vs D-MOFARC	43	48	0.83	**Not Rejected**

from the curse of dimensionality: the higher the number of input variables, the more difficult the generation of the set of candidate rules. In addition, the last step of FARC-HD and D-MOFARC requires the execution of an evolutionary algorithm for selecting a reduced set of rules. As stated in [13], the size of the search space grows with the increase of the number of input variables, thus leading to a slow and possibly difficult convergence of the evolutionary algorithm. Further, the computational cost of the fitness evaluation increases linearly with the increase of the number of instances in the dataset, thus obliging to limit the number of evaluations especially when the dataset is large. On the other hand, AC-FFP needs only four scans of the dataset.

3.5 DAC-FFP: a Distributed implementation of AC-FFP for Big Data

In Section 3.4, we have pointed out that the use of fuzzy partitions makes the fuzzy CAR mining more complex. Indeed, while in the case of crisp partitions an input value supports a unique item, in the case of fuzzy partitions, an input value can support more than one fuzzy item (in our implementation, which is based on strong fuzzy partitions, the supported items are at most two). Thus, the number of possible fuzzy association rules is higher than the number of possible crisp rules. This issue is much more evident when dealing with big datasets.

So far, the AC-FFP algorithm described in Section 3.4 is not able to deal with both large and high dimensional datasets. Although the use of the FP-Growth algorithm guarantees a higher level of efficiency in terms of memory occupation and computational time than the Apriori algorithm, this advantage is not enough to manage big data. Indeed, the high dimensionality might generate very large FP-trees, which cannot be contained in the main memory. Further, the size of the dataset could preclude storing the overall amount of data in the main memory. Both these drawbacks require frequent swapping operations, which dramatically affect the computational time of AC-FFP method. Moreover, the classification accuracy achieved by AC-FFP, although comparable to the ones obtained by similar state-of-the-art algorithms proposed recently in the literature, might be improved by employing a different discretization approach. Indeed, the accuracy depends on how the continuous attributes are partitioned. Thus, fuzzy discretization approaches, which directly generate optimal fuzzy partitions rather than fuzzifying discrete partitions are more desirable.

With this aim, we propose a novel distributed fuzzy partitioning for generating Ruspini fuzzy partitions on each continuous attribute and an efficient fuzzy associative classifica-

tion scheme for dealing with big data. The proposed approach, based on the MapReduce paradigm, is a distributed version of our Associative Classifier based on a Fuzzy Frequent Pattern (AC-FFP) mining algorithm. The overall algorithm, denoted as DAC-FFP, first distributes the generation of Ruspini fuzzy partitions by exploiting the fuzzy entropy, then mines fuzzy associative classification rules by employing a distributed implementation of the fuzzy frequent pattern mining algorithm proposed in AC-FFP and finally selects a set of fuzzy CARs by performing different distributed pruning steps. For handling big dataset and improving the accuracy of the fuzzy classifier as well, we introduce several strategies that improve the performance of DAC-FFP. Indeed, we propose: i) a novel fuzzy discretization step for generating Ruspini fuzzy partitions, ii) an enhanced fuzzy rules mining approach for speeding up the execution time similar to the one employed by MRAC, and iii) an ad-hoc classification method for improving the accuracy of the classifier. Referring to an implementation on Hadoop, we show memory usage and time complexity for each phase of the learning process.

We adopt seven real-world big datasets with different numbers of instances (up to 11 millions) to analyze scalability and speedup of the algorithm according to different work units. Further, focusing on accuracy, model complexity and computation time, we compare the results obtained by our approach with those achieved by two crisp associative classifiers, namely MRAC and MRAC+, described in Section 3.2.

The rest of the section is organized as follows. Section 3.5.1 describes the proposed approach, with details of each single job that runs on the cluster of machines. We exploit Hadoop for the implementation of our distributed fuzzy associative classifier. Section 3.5.2 presents the experimental setup and discusses the results in terms of accuracy, computation time, model complexity, speedup, and scalability.

3.5.1 The Distributed Approach

The overall FCAR mining algorithm, that allow us to generate a classical FRBC, can be summarized into three main stages:

1. *Distributed Fuzzy Partitioning*: a strong fuzzy partition is defined on each continuous attribute using a distributed approach based on fuzzy entropy;
2. *Distributed Fuzzy CAR Mining*: a distributed fuzzy frequent pattern mining algorithm is exploited to extract frequent fuzzy classification rules with confidence and support higher than a pre-fixed threshold. The implementation of this stage is based the parallel FP-Growth algorithm discussed in [106].
3. *Distributed Fuzzy CAR Pruning*: the mined CARs are pruned by means of two distributed rule pruning phases based on redundancy and training set coverage. The remaining CARs are kept in the final RB employed for classifying unlabeled instances.

In the following, we first detail the three stages introduced above. Moreover, we will also discuss how the classification of unlabeled instances can be carried out by employing the generated FRBC model. For sake of clarity, we refer to its implementation upon

Hadoop. The overall process is carried out by performing five MapReduce phases that involve four training set scans.

Distributed Fuzzy Partitioning

As stated before, granularity learning is a critical aspect in the generation of fuzzy association rule, and in general in the design of FRBCs. Indeed, the performance of classifiers can be significantly affected by the different methodologies that can be employed for both continuous attribute discretization and fuzzy set parameters definition. With this aim, we propose a novel fuzzy partitioning approach based on fuzzy entropy for generating strong triangular fuzzy partitions on each continuous feature. In particular, the algorithm is a supervised method that, recursively, determines the core of each triangular fuzzy set by exploiting the class information fuzzy entropy of each candidate partition.

The algorithm follows a top-down approach working on intervals. In particular, at each stage, the algorithm aims at identifying, in the considered interval $I_{f,s}$ of the variable X_f, the parameters of a strong fuzzy partitions built using three triangular membership functions. Let $S_{I_{f,s}} = [x_{1,f}, ..., x_{N_{I_{f,s}},f}]^T$ be the set of $N_{I_{f,s}}$ points, in the interval $I_{f,s}$, of the projection of the training set TS along variable. Let $l_{f,s}$ and $u_{f,s}$ be the lower and upper bounds of $I_{f,s}$. Let $t_{f,s}$ be a point in $S_{I_{f,s}}$, we define a *candidate fuzzy partition* in $I_{f,s}$ by using three triangular fuzzy sets, namely $A_{I_{f,s},1}$, $A_{I_{f,s},2}$, $A_{I_{f,s},3}$. Indeed, the cores of $A_{I_{f,s},1}$, $A_{I_{f,s},2}$, $A_{I_{f,s},3}$ coincide with $l_{f,s}$, $t_{f,s}$, and $u_{f,s}$, respectively.

Let S_1, S_2 and S_3 be the subsets of points in $S_{I_{f,s}}$ which lie in the support of $A_{I_{f,s},1}$, $A_{I_{f,s},2}$ and $A_{I_{f,s},3}$, respectively. The class information fuzzy entropy induced by $t_{f,s}$ and denoted as $WFEnt(t_{f,s}; S_{I_{f,s}})$ is computed by exploiting the weighted fuzzy entropy:

$$WFEnt(t_{f,s}; S_{I_{f,s}}) = \sum_{j=1}^{3} \frac{|S_j|}{|S|} FEnt(S_j) \tag{3.33}$$

where $|S_j|$ and $|S|$ are the fuzzy cardinalities of subset S_j and S respectively, and $FEnt(S_j)$ is the fuzzy entropy of S_j.

We recall that the fuzzy cardinality of a subset of points S_j, which lie in the support of the fuzzy set $A_{I_{f,s},j}$, is computed as

$$|S_j| = \sum_{i=1}^{N_j} \mu_{S_j}(\mathbf{x}_i) = \sum_{i=1}^{N_j} \mu_{A_{I_{f,s},j}}(\mathbf{x}_i) \tag{3.34}$$

where N_j is the number of points (crisp cardinality) in the subset S_j, $\mu_{S_j}(\mathbf{x}_i) = \mu_{A_{I_{f,s},j}}(\mathbf{x}_i)$ is the membership degree of example $\mathbf{x}_i$ to subset S_j and $\mu_{A_{I_{f,s},j}}(\mathbf{x}_i)$ is the membership degree of example $\mathbf{x}_i$ to fuzzy set $A_{I_{f,s},j}$. On the other hand, the fuzzy entropy of S_j is defined as

$$FEnt(S_j) = \sum_{l=1}^{L} -\frac{|S_{j,C_l}|}{|S_j|} log_2(\frac{|S_{j,C_l}|}{|S_j|}) \tag{3.35}$$

where S_{j,C_l} is the set of examples in S_j with class label equal to C_l and L is the total number of class labels.

The optimal point $t_{f,s}^{min}$, which minimizes the class information fuzzy entropy over all possible candidate fuzzy partitions in $I_{f,s}$, is selected as actual core of the central fuzzy set of the strong partition.

The proposed fuzzy partitioning method starts considering, for each input variable X_f, an initial interval equal to the whole variable domain, namely $I_{f,0} = U_f$. The process is then recursively applied to both intervals $I_{f,1}$ and $I_{f,2}$ identified by $t_{f,0}^{min}$. Given a specific interval $I_{f,s}$, the process stops if the following criterion is achieved:

$$FGain(t_{f,s}^{min}; S_{I_{f,s}}) < \frac{\log_2(|S_{I_{f,s}}| - 1)}{|S_{I_{f,s}}|} + \frac{\Delta(t_{f,s}^{min}; S_{I_{f,s}})}{|S_{I_{f,s}}|} \tag{3.36}$$

where

$$FGain(t_{f,s}^{min}; S_{I_{f,s}}) = FEnt(S_{I_{f,s}}) - WFEnt(t_{f,s}^{min}; S_{I_{f,s}}), \tag{3.37}$$

$$\Delta(t_{f,s}^{min}; S_{I_{f,s}}) = \log_2(3^L - 2) - \left[L \cdot FEnt(S_{I_{f,s}}) - \sum_{j=1}^{3} L_j \cdot FEnt(S_j)\right] \tag{3.38}$$

and L_j is the number of class labels represented in the set S_j.

At the end of overall process, the algorithm identifies a strong fuzzy partition P_f for each input variable X_f. If no MFs have been defined by the algorithm, X_f is discarded and not employed in the rule mining process. This situation can happen whenever, during the exploration of $I_{f,0}$, formula (3.36) is not verified for any candidate central core $t_{f,0}$.

The complexity of the algorithm depends on the number of candidate fuzzy partitions evaluated in each interval $I_{f,s}$. Indeed, to determine the best $t_{f,s}^{min}$, the algorithm first should sort all the unique values of the variable, and then defines a candidate fuzzy partition on each unique value. However, this approach is not suitable to deal with a huge amount of data because sorting values and checking the candidate partitions could be a computational expensive task in case of very large distributed datasets that involves millions or even billion of points. To overcome this drawback, we adopt a kind of undersampling, in which the set of instances, in the projection of the training set along each input variable X_f, is approximated with a reduced number of representative points. To this aim, we perform a equi-frequency binning of each variable and identify the boundaries of each bin. Such boundaries are labeled as the representative points of the specific variable X_f. During the fuzzy partition identification process, bin boundaries which lie in a specific interval $I_{f,s}$ are analyzed as candidate central cores. To this aim, the class distribution for each bins is calculated: we considered the number of actual instances belonging to the different classes in each bins. Note that the calculation of the class distributions is carried out just once.

For handling big dataset, we propose a distributed implementation of the previously discussed fuzzy partitioning method under the MapReduce paradigm. The proposed distributed implementation also includes the undersampling process for the generation of the representative points of the projection of the training set along each input variable.

In Figure 3.21, we show the employed architecture, which highlights the implemented Map and Reduce functions. We consider that Z is the number of blocks used for splitting the training set and Q is the number of Computing Units (CUs) available across the cluster. Each block fed only one *Map-Task* and of course one CU can process several tasks, both Map or Reduce. Obviously, only Q tasks can be executed in parallel.

The overall distributed fuzzy partitioning process is carried out by using two MapReduce phases. In the first MapReduce step, the algorithm splits each continuous variable into several bins. First in the map phase, each CU loads a block of training set and then computes the bin boundaries by splitting the data according to a fixed number of equi-frequency bins. Then, in the reduce phase, for each variable all the bin boundaries are grouped together and then each CU generates a sorted list of bin boundaries for each variable.

The second MapReduce step carries out the fuzzy partitioning of each variable, considering the bin boundaries as representatives of the actual instances. In the map phase, each CU computes the number of instances belonging to the different classes for each bin determined by a pair of consecutive bin boundaries in the list. In the reduce phase, each CU generates, for each input variable, a strong fuzzy partition employing the fuzzy partitioning method previously discussed. In particular, the bin boundaries generated during the first MapReduce stage are considered as candidate central cores. The fuzzy entropy is always calculated by using the distribution of classes in each interval identified by two consecutive bin boundaries, previously calculated by the mappers.

The proposed distributed approach makes it feasible to handle a large number of objects: the bin boundaries allow to reduce the data amount on which the Fuzzy Partitioning is applied to. Obviously, the higher the frequency used in the equi-frequency bins, the coarser the approximation in determining the best fuzzy partition. However, we must notice that, we are managing millions of data, thus a difference of a few instances between bin boundaries instead of the original instances is generally negligible in terms of achieved accuracy.

The first MapReduce phase scans the training set to compute at most $\Omega = Z \cdot \Gamma$ bin boundaries, where $\Gamma = 100/\gamma + 1$ is the number of bin boundaries per block. This value depends on the percentage γ of the zth block size. In our experiments, we set $\gamma = 0.1\%$. Each Map-Task, first, loads the zth block of the training set, and then for each variable X_f , computes and outputs the bin boundaries of equi-frequency bins, where each bin contains a number of instances equal to the percentage γ of the data block. Let $B_{z,f} = \{b_{z,f}^{(1)}, ..., b_{z,f}^{(\Gamma)}\}$ be the sorted list of bin boundaries for the fth variable extracted from the zth block, the Map-Task outputs a key-value pair $\langle key = f, value = B_{z,f} \rangle$ where f is the index of the fth variable. Each Reduce-Task is fed by Z lists, say $List(B_{z,f})$, and for the fth variable, it outputs $\langle key = f, value = B_f \rangle$, where $B_f = \{b_f^{(1)}, ..., b_f^{(\Omega)}\}$ with $\forall w \in [1, ..., \Omega - 1]$ $b_f^{(w)} < b_f^{(w+1)}$ is the sorted list of the bin boundaries for the variable X_f. Space and time complexities, for the Map phase, are $O(\lceil \frac{Z}{Q} \rceil \cdot N/Z)$ and $O(\lceil \frac{Z}{Q} \rceil \cdot (F \cdot N \cdot (log(N/Z))/Z))$. For the Reduce phase, space and time complexities are $O(F \cdot \Omega/Q)$ and $O(F \cdot (\Omega \cdot log(\Omega))/Q)$, respectively.

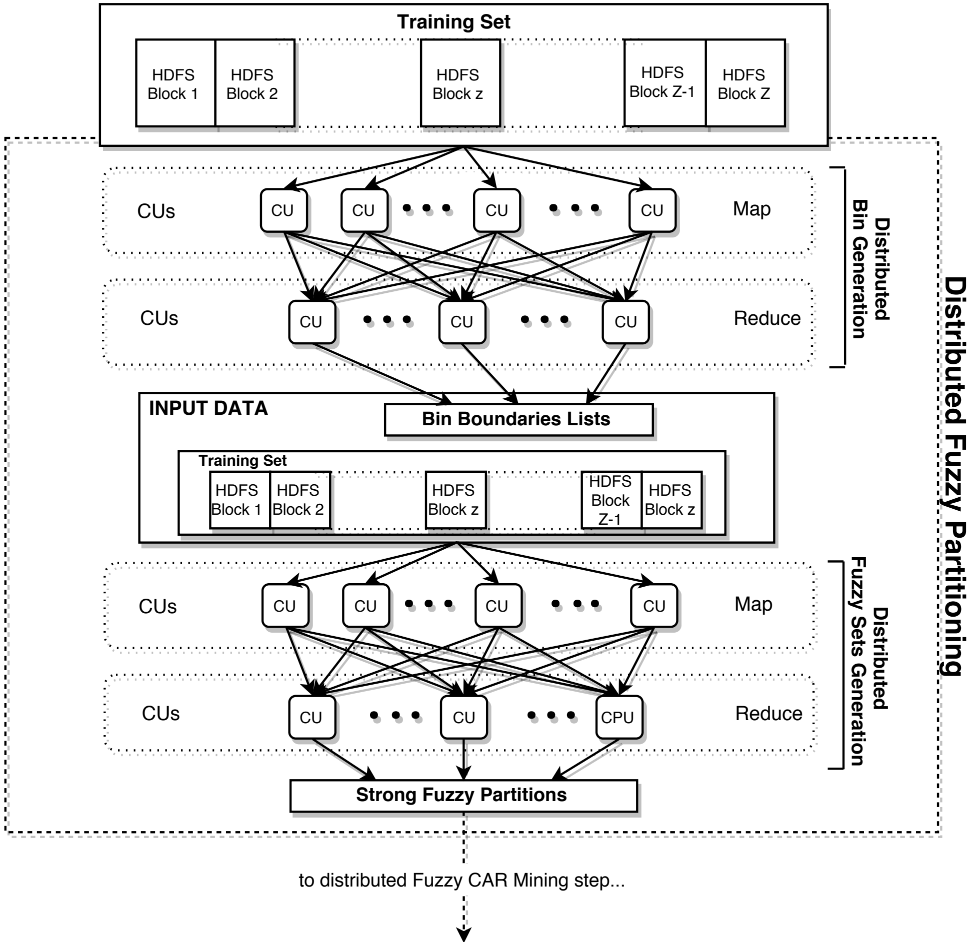


Figure 3.21: The overall distributed Fuzzy Partitioning of the Fuzzy Decision Tree.

As stated before, the second MapReduce phase aims at generating the most suitable fuzzy partition for each variable X_f: the map phase computes the number of actual instances of each class for each bin, and the reduce phase performs the identification of the best fuzzy partitioning for each variable X_f. Each Map-Task, first, loads the zth block of the training set and for each input variable X_f, initializes a vector $W_{z,f}$ of $\Omega - 1$ elements. Each element $W_{z,f}^{(r)}$ corresponds to the bin $(b_f^r, b_f^{(r+1)}]$ and contains a vector of L elements, which stores, for each of the L classes, the number of instances of the class belonging to the rth bin in the zth block. Then, for each object of the block, the Map-Task updates $W_{z,f}$ and finally outputs a key-value pair $\langle key = f, value = W_{z,f} \rangle$. Each Reduce-Task is fed by a list, say $List(W_{z,f})$, of Z of $W_{z,f}$ vectors and for each variable X_f, it first creates a vector W_f of $\Omega - 1$ elements by performing an element-wise addition of all Z vectors $W_{z,f}$. Thus, W_f stores the frequency of each class in every bin along

the whole training set. Then, the Reduce-Task applies the fuzzy partitioning method as described above, where the *candidate fuzzy partitions* are defined upon bin boundaries and the fuzzy mutual information is computed according to W_f. Finally, each reducer outputs the key-pair $\langle key = f, value = P_f \rangle$, where P_f is the strong fuzzy partition defined on the variable X_f. Space and time complexities of the Map phase are $O(\lceil \frac{Z}{Q} \rceil \cdot N/Z)$ and $O(\lceil \frac{Z}{Q} \rceil \cdot (N \cdot log(\Omega)/Z))$, respectively. For Reduce phase, space and time complexities are $O(F \cdot (\Omega - 1)/Q)$ and $O(F \cdot max(FuzzyPart(X_f))/Q)$, respectively, where $FuzzyPart(X_f)$ is the time complexity of the fuzzy partitioning method for retrieving the best central core in each interval $I_{f,s}$ of the fth variable. Since for each variable, the fuzzy partitioning method generates a different number of intervals, the time complexity is upper bounded by the maximum value $max(FuzzyPart(X_f))$.

Distributed Fuzzy CAR Mining

The distributed fuzzy CAR mining approach extracts, for each class label, K non-redundant fuzzy association rules, characterized by the highest confidence, by performing two scans of the overall training set. The implementation is based on the Parallel FP-Growth (PFP-Growth) proposed by Li et al. [106] for efficiently distributing the frequent patterns mining without generating candidate item sets. First, the algorithm selects the frequent items, builds the f_{list} and then distributes *item-projected datasets* on each node for building local and independent FP-Trees. An *item-projected dataset* $T(IT_{f,j})$ contains only objects, also called *item-projected objets*, where the items are sorted according to the f_{list} and the items with lower support than $IT_{f,j}$ are removed. In the last phase, the algorithm aggregates the results and, for each item, selects the highest supported patterns.

We adapted the PFP-Growth algorithm to generate frequent FCARs characterized by a high fuzzy confidence. As shown in Figure 3.22, the overall fuzzy FCAR mining process is carried out by performing three MapReduce phases: (i) *parallel fuzzy counting*, (ii) *parallel fuzzy FP-Growth*, and (iii) *parallel rules selection*.

The *parallel fuzzy counting* phase scans the dataset and counts both the fuzzy support for selecting the *frequent fuzzy sets*, and the number of occurrences of each class label. A fuzzy set is frequent if its fuzzy support is higher than a minimum threshold *minSup* fixed by the expert. The MapReduce framework divides the entire training set into blocks and assigns each of them to a map task. Each map task is fed by input key-value pairs represented as $\langle key = r, value = \mathbf{o}_r \rangle$, where $\mathbf{o}_r = (\mathbf{x}_r, y_r)$ is the rth object of the training set block. For each fuzzy set $A_{f,j} \in P_f$, the mapper outputs a key-value pair $\langle key = A_{f,j}, value = A_{f,j}(x_{r,f}) \rangle$, where $A_{f,j}(x_{r,f})$ is the membership degree of the f^{fh} component of $\mathbf{x}_r$ to the jth fuzzy set of partition P_f. Obviously, only the fuzzy sets with matching degree higher than zero are considered. Since we use strong partitions, for each $\mathbf{x}_r$, we output at most $2F$ values. Finally, the mapper outputs also the key-value pair $\langle key = y_r, value = 1 \rangle$. The reducer is fed by a list of corresponding values for each key: a set of membership degrees for

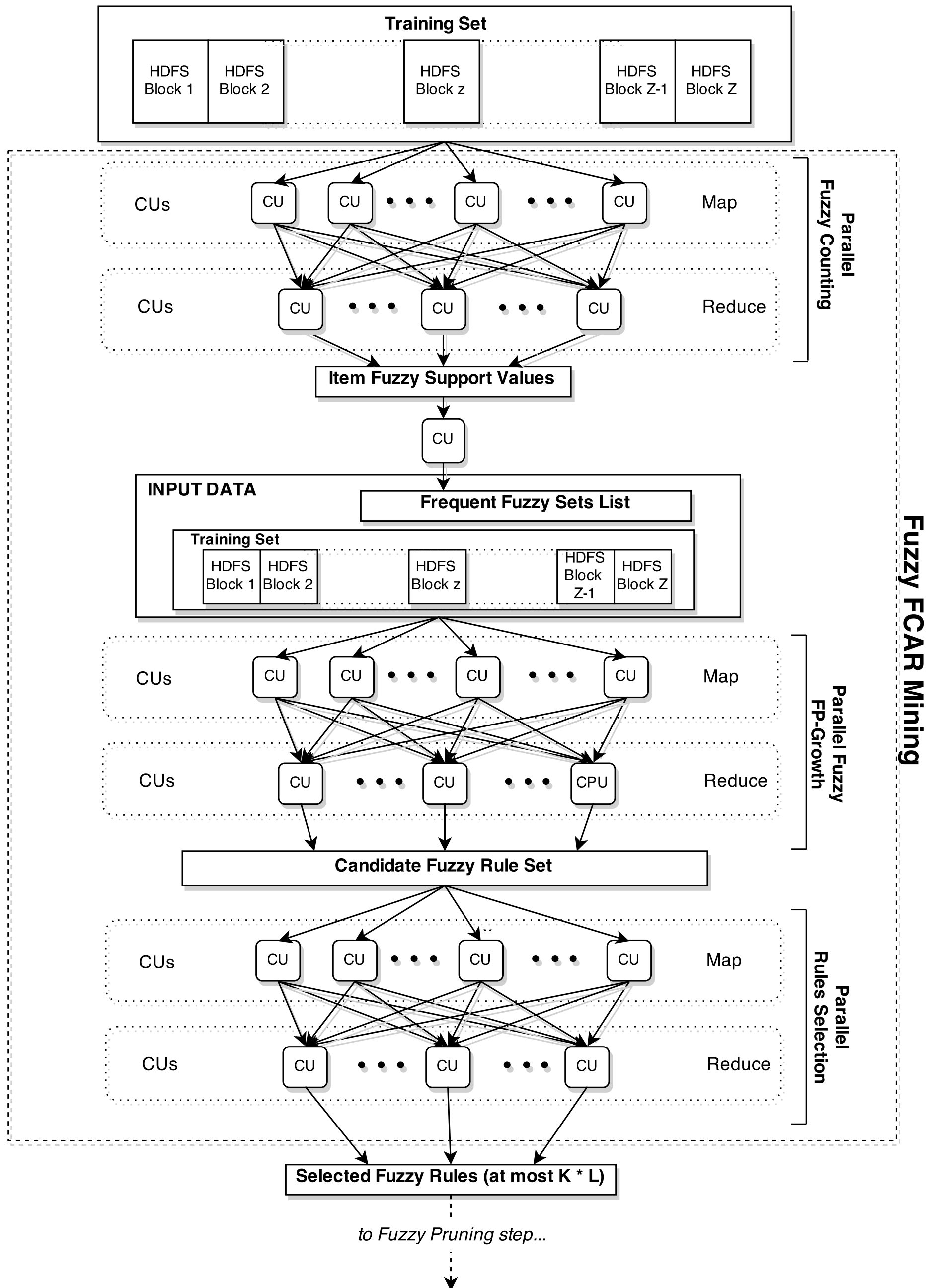


Figure 3.22: The overall FCAR Mining process of the DAC-FFP algorithm.

the fuzzy sets, and a set of 1's for the class labels. The reducer input is formatted as $\langle key = A_{f,j}, value = list(A_{f,j})\rangle$ and $\langle key = C_j, value = list(C_j)\rangle$, respectively, and outputs $\langle key = A_{f,j}, value = fuzzySupp(A_{f,j}))\rangle$, where $fuzzySupp(A_{f,j})$ is calculated according to the Eq. 3.29, and $\langle key = C_j, value = size(list(C_j))\rangle$. Space and time complexity are both O(N/Q), where Q is the number of CUs.

At the end of the first MapReduce phase, the algorithm selects only the fuzzy sets whose support is larger than the support threshold $minSup$, stores them in the f_{list}, sorting in descending order according to the fuzzy support, and prunes the other ones. Only the frequent fuzzy sets will be considered in the subsequent phases. Since f_{list} is generally small, this step can efficiently be performed on a single machine.

The second MapReduce phase, *parallel fuzzy FP-Growth*, mines fuzzy CARs whose support, confidence and χ^2 values are higher than *minSup*, *minConf* and $min\chi^2$ thresholds, respectively. The approach is very similar to the classical parallel FP-Growth described by Li et al. [106] with the only difference that we extend the projected dataset and projected object concepts to the fuzzy context. Indeed, each mapper computes the *item projected objects* so that the reducers are able to build the *item-projected datasets*, and then to generate local conditional FP-Trees. We recall that in our case an item is defined as $IT_{f,j} = (X_f, A_{f,j})$. In the following we denote the $A_{f,j}$*-projected dataset* as $T(A_{f,j})$. Since, the local conditional FP-Trees are independent of each other, the fuzzy CARs can be mined by each node independently of the other nodes.

As in the first phase, each mapper reads a key-value pair $\langle key = r, value = \mathbf{o}_r\rangle$ in the training set block, and then builds the fuzzy object $\widetilde{\mathbf{o}}_r$ from $\mathbf{o}_r$. As described in Section 3.4.1, for each $x_{r,f}$ value, the mapper extracts the fuzzy sets $A_{f,j}$ with the highest matching degree from the two fuzzy sets activated by $x_{r,f}$. Then, the mapper sorts the fuzzy sets according to the f_{list}. If a fuzzy set is not present in the f_{list} then it is discarded. Finally, the mapper retrieves the class label C_{j_n} and, for each extracted fuzzy set, outputs the key-value pair $\langle key = index_{A_{f,j}}, value = A_{f,j} - projected\, object\rangle$, where $index_{A_{f,j}}$ is the index of the fuzzy set $A_{f,j}$ in the f_{list}.

The MapReduce framework groups all the item-projected objects with the same index, and passes them to the reducer. Each reducer is able to process the $A_{f,j}$-projected training sets independently, and generates the associated FCARs. Indeed, the reducer receives in input key-value pairs $\langle key = index_{A_{f,j}}, value = T(A_{f,j})\rangle$, builds the local conditional FP-Tree and recursively mines the FCARs, as described in [80]. As in the AC-FFP, if a node already exists in the tree, the corresponding counter is simply incremented by 1 and no other information about the matching degrees are maintained. Finally, when rules are extracted from the FP-Tree, only those FCARs whose support, confidence and χ^2 values are greater than the relative thresholds are considered. In particular, reducers output $\langle key = null, value = FCAR_m\rangle$ pairs, where $FCAR_m$ is the mth generated rule. Space and time complexities of each reducer depend on the size and the execution time of all the processed projected training sets, $O_{red}(Sum(|T(v_{f,j})|))$ and $O_{red}(Sum(FPGrowth(T(v_{f,j}))))$, respectively.

The recursive mining of FCARs from the local FP-tree can be very hard from the computational point of view, especially when dealing with Big Data. Moreover, handling a huge number of instances often leads to generating a large number of FCARs. Similar to MRAC, We designed a methodology to speedup the FCAR mining process by reducing the number of generated FCARs. In order to avoid the inspection of several sub-trees, and thus the generation of several FCARs, we adopt a sort of rule pre-pruning during the FCAR mining process. More precisely, the method avoid to inspect sub-trees that will likely generate redundant rules. In the following, we introduce some additional definitions is order to give the precise explanation of the pre-pruning step. A $FCAR_m$ is more significant than another $FCAR_s$ if and only if:

1. $conf(FCAR_m) > conf(FCAR_s)$
2. $conf(FCAR_m) = conf(FCAR_s)$ **AND** $supp(FCAR_m) > supp(FCAR_s)$;
3. $conf(FCAR_m) = conf(FCAR_s)$ **AND** $supp(FCAR_m) = supp(FCAR_s)$ **AND** $RL(FCAR_m) < RL(FCAR_s)$.

where $conf(.)$, $supp(.)$, and $RL(.)$ are the confidence, the support and the rule length respectively. A fuzzy rule $FCAR_s$ is pruned if and only if exists a rule $FCAR_m$ with higher rank and more general than $FCAR_s$. A rule $FCAR_m : FAnt_m \rightarrow C_{j_m}$ is more general than a rule $FCAR_s : FAnt_s \rightarrow C_{j_s}$, if and only if, $FAnt_m \subseteq FAnt_s$. For each of these K rules the reducer outputs a key-value pair $\langle key = null, value = FCAR_m \rangle$. This lets us discard redundant rules and cover a larger number of objects in the training set. The rules are more and more specialized as the recursive visit of a $A_{f,j}$-conditional FP-tree goes deeper in the tree. We stop the visit of an FP-tree at a specific node ι if, visiting two further nodes, the rules generated by adding the items corresponding to the nodes are not more significant than the rule generated at node ι. Let us assume that $FCAR_m : FAnt_m \rightarrow C_{l_m}$, $FCAR_s : FAnt_s \rightarrow C_{l_s}$, and $FCAR_d : FAnt_d \rightarrow C_{l_d}$, with $FAnt_s = \{FAnt_m, \widetilde{A}_{f,j}\}$ and $FAnt_d = \{FAnt_s, \widehat{A}_{f,j}\}$, where $\widetilde{A}_{f,j}$ and $\widehat{A}_{f,j}$ are frequent items for the $FAnt_s$ and $FAnt_d$ antecedents, respectively. We stop the generation of other rules with sub-pattern $FAnt_d$ if and only if:

$$\begin{aligned} conf(FCAR_d) - conf(FCAR_s) \leq 1 - \tfrac{supp(FCAR_d)}{supp(FCAR_s)} \\ AND \\ conf(FCAR_s) - conf(FCAR_m) \leq 1 - \tfrac{supp(FCAR_s)}{supp(FCAR_m)} \end{aligned} \tag{3.39}$$

The pre-pruning method does not evaluate Eq. 3.39 for the rules with one or two antecedents and for the rules generated from $FAnt_d$ whose class is different from $FCAR_d$. In this way, we avoid to visit the paths in the FP-tree that probably will generate redundant rules. We recall that such paths would be pruned in the next steps anyway. Thanks to this modification of the FP-Growth, we can significantly reduce the execution time of the mining process and, thus, we can reduce the value of the $minSup$ threshold. In this way, the mining process generates highly confident and specialized rules with a small support. In our experiments, we verified that this pre-pruning allows us to obtain a good trade-off between execution time and accuracy performance of the generated classifiers.

Note that the number of pairs $\langle key, list(values) \rangle$ processed by each reducer is determined by the default partition function $hash(key)$ **mod** R. Since in *parallel Fuzzy FP-Growth* the intermediate key is the specific item index in f_{list}, more or less the same number of pairs $\langle key, list(values) \rangle$, i.e. of conditional FP-trees, is assigned to each reducer by the partition function. However, such a distribution does not necessarily guarantee a perfect load balancing among all the reducers, because the time spent in processing each specific conditional FP-tree depends on the number and length of its paths; more precisely, the relative time complexity is exponential with respect to the longest frequent path in the conditional pattern base [152, 212]. In case of a large number of frequent items, the conditional FP-trees corresponding to the items with the smallest support are very deep since they consider in their paths almost all the frequent items. The rule pre-pruning can reduce this problem for specific datasets, but cannot solve it in general. Thus, datasets whose objects are described by a small number of features can be easily managed, but conversely runtime problems may occur in dealing with objects with a large number of features.

Note that the MapReduce framework automatically determines the number of conditional FP-Trees assigned to each reducer through a default partition function, namely $hash(key)\ mod\ R$, where R is the number of reducers. Even though our implementation ensures more or less the same number of conditional FP-Trees processed by each reducer, such distribution does not necessarily guarantee a perfect load balancing among all the nodes, because the time spent in processing each specific FP-Tree depends on the number and length of its path [152, 212].

Since the number of the rules generated in the previous step can be very high, the next MapReduce phase, *parallel rules selection*, selects only the top K non-redundant rules for each class label C_j. Each mapper is fed by an input key-value pair formatted as $\langle key = m, value = FCAR_m \rangle$ and outputs a pair $\langle key = C_{j_m}, value = FCAR_m \rangle$, where C_{j_m} is the class label associated with $FCAR_m$. Each reducer processes all rules with the same class label, $List(FCAR_{C_l})$ and selects the best K significant non-redundant rules. For each of these K rules the reducer outputs a key-value pair $\langle key = null, value = FCAR_m \rangle$. Time complexity is $O(Max(|FCAR_{C_l}|) \cdot log(K)/Q)$, where Q is the number of CUs.

Distributed Fuzzy CAR Pruning

Figure 3.23 shows the MapReduce phases which are in charge of executing the pruning of the mined FCAR rules, with the main aim of reducing noisy information. At this stage, two additional scans of the overall training set are carried out.

Given an $FCAR_m$ rule, during the first scan, the algorithm verify if the fuzzy support and confidence are lower than a $minFuzzySupp$ and $minFuzzyConf$ thresholds, calculated according to formulas 3.30 and 3.31, respectively. If the both conditions are both verified, rule $FCAR_m$ is discarded.

The mapper is fed by the training set block where each input takes the form $\langle key = r, value = \mathbf{o}_r \rangle$. Moreover, each mapper loads and ranks the rules mined in the previ-

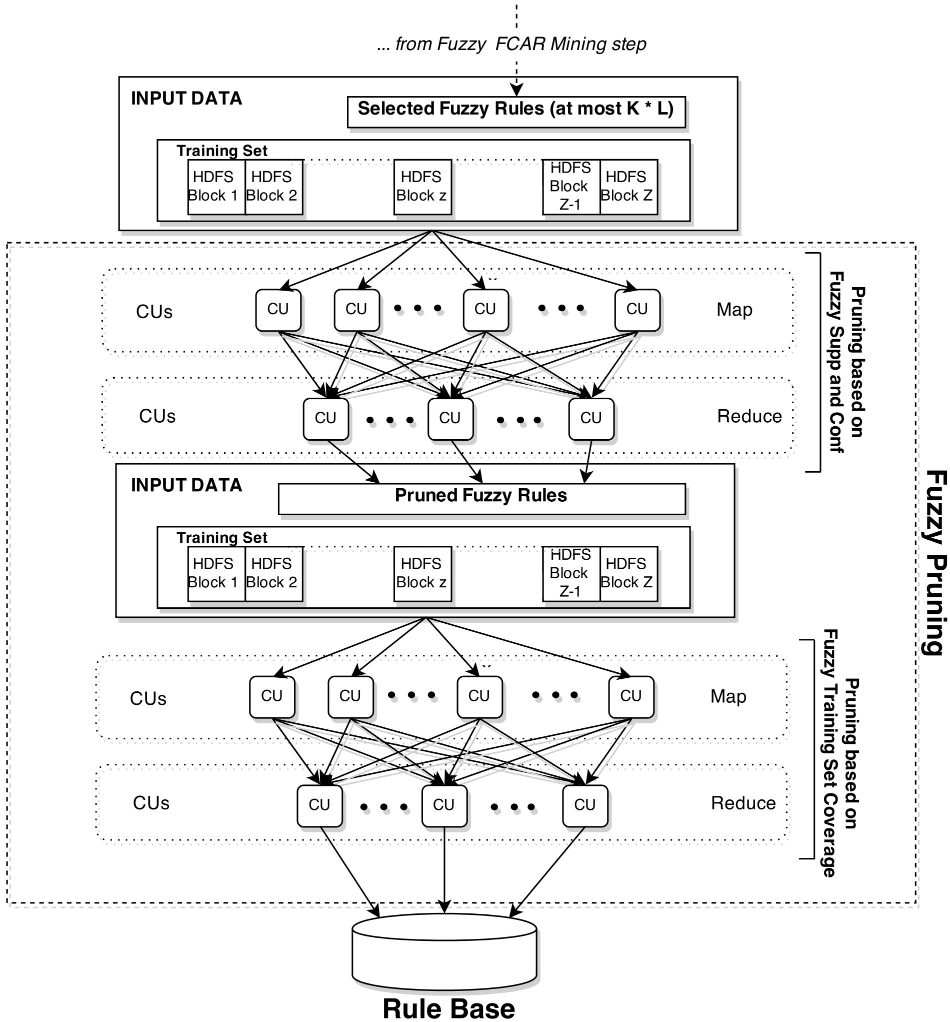


Figure 3.23: The overall FCAR Pruning process of the DAC-FFP algorithm.

ous steps into a ranked list and calculates the membership degree of each rule for $\mathbf{o}_r$ according to Eq. 3.15, by using the product as t-norm for implementing the conjunction operator. For each $FCAR_m$ with membership degree higher than zero, the mapper outputs the key-value pair $\langle key = index_{FCAR_m}, value = w_m(\mathbf{x}_r)\rangle$, where the key is the index of rule $FCAR_m$ in the ranked list and $w_m(\mathbf{x}_r)$ is the membership degree of rule $FCAR_m$. Note that, when the class label C_{j_m} of rule $FCAR_m$ is different from class label y_r of object $\mathbf{o_r}$, the mapper outputs a negative value, $-w_m(\mathbf{x}_r)$, thus the reducer is able to properly compute the fuzzy support and the fuzzy confidence. Indeed, each reducer inputs a key-value pair as $\langle key = index_{FCAR_m}, value = list(w_m)\rangle$, where $list(w_m)$ is the list of all non-zero matching degrees for the rule $FCAR_m$. The fuzzy

support and the fuzzy confidence are calculated according to Equations 3.23 and 3.24, where the numerators of both equations is computed by considering only the positive w_m, and the denominator of the second one is computed by considering the absolute value of each element in the $list(w_m)$. Finally, each reducer outputs only FCARs with confidence and support higher than $minFuzzySupp$ and $minFuzzyConf$, respectively $\langle key = null, value = FCAR_m \rangle$. Space complexity is $O(N/Q)$ and time complexity is $O(N \cdot |CAR_{list}|/Q)$, where $|CARlist| \leq L \cdot K$ and Q is the number of CUs.

A training set coverage analysis is carried out as the last step of the pruning process. The training set blocks are used to feed the mappers and $\langle key = r, value = \mathbf{o}_r \rangle$ is used as key-value input pair. Further, the filtered rule set generated in the previous phase is loaded and ranked by each mapper. As in the AC-FFP, a counter initialized to 0 is associated with each object $\mathbf{o_r}$. The mapper scans the rule list and, for each $FCAR_m$, if $\mathbf{x}_r$ matches the rule, then the counter is incremented by 1. Only those rules with matching degree higher than the *fuzzy matching degree threshold* $\overline{w}_m = 0.5^{g_m - 1}$, where g_m is the rule length of $FCAR_m$, are considered. If the $FCAR_m$ correctly classifies $\mathbf{x}_r$, the mapper outputs a key-value pair as $\langle key = index_{FCAR_m}, value = null \rangle$, where the key is the index of the $FCAR_m$ in the ranked list. When the counter is higher than a *coverage threshold* δ, the corresponding object is not considered anymore. The key-value pair $\langle key = index_{FCAR_m}, value = null \rangle$ is used to feed the reducer, which gets the rule from the ranked list and outputs $\langle key = null, value = FCAR_m \rangle$. Space complexity is $O(N/Q)$ and time complexity in the worst case is $O(N \cdot |CAR_{list}|/Q)$, where $|CARlist| \leq L \cdot K$ and Q is the number of CUs.

Reasoning methods

At the end of the previously discussed methodology for generating fuzzy associative classifiers, we obtain a complete knowledge base which in include both the final rule base and data base. As stated in Section 3.3.1, on the basis of the knowledge base, a specific reasoning method is employed to classify unlabeled patterns. In the experiments that we will discuss in the following, we employ both the *weighted vote* and the *maximum matching* reasoning methods. Since we adopt the product t-norm as conjunction operator, rules with a higher number of conditions in the antecedent have generally a lower matching degree than rules with a lower number of conditions in the antecedent. Hence, more general rules are more influential than specific rules in the prediction phase. In order to re-introduce a balance on the influence of the rules, for a given input pattern $\widehat{\mathbf{x}} = [\hat{x}_1 \ldots, \hat{x}_F]$ and a given rule $FCAR_m$, we normalize the matching degree of rules as follows:

$$\widehat{w}_m(\widehat{\mathbf{x}}) = w_m(\widehat{\mathbf{x}}) \cdot 2^{g_m} \tag{3.40}$$

where $\widehat{w}_m(\widehat{\mathbf{x}})$ and $w_m(\widehat{\mathbf{x}})$ are the normalized matching degree and the matching degree of $FCAR_m$ for the input $\widehat{\mathbf{x}}$, respectively, and g_m is the number of antecedent conditions of $FCAR_m$. Note that in case $\widehat{\mathbf{x}}$ activates no rules, all the reasoning methods classify the input pattern as *unknown*.

The quality of fuzzy association rules is mainly driven by the value of its confidence. Even though both maximum matching and weighted vote methods employ such value as certainty factor, rules with high confidence are not properly rewarded. For instance, maximum matching tends to select rules that are characterized by higher association degrees and weighted vote consider the contribution of each rule, taking care also about those rules with low confidence. On the other hand, a rule with a confidence equals to 100% should be always selected for classifying the input pattern, independently of the vote or the activation degree of other rules. Such kind of rules are very representative of training dataset because every time they have been activated during the model learning process, they have been able to classify the pattern with the right class label.

With the aim to improve the performance of classifier, we also experimented an additional reasoning method carefully designed for fuzzy associative classifiers. First, we rank the rule base so that all rules are sorted according to confidence, support and number of antecedent conditions. Then, given an unlabeled pattern $\widehat{\mathbf{x}}$, we output the class label $\widehat{C}_l$ corresponding to the first rule with $\widehat{w}_m(\widehat{\mathbf{x}})$ greater than a pre-fixed threshold. We denoted this novel reasoning method as *best rule*. In our experiments, we set the threshold to 1 so that a rule is considered only when in average the membership degree of each fuzzy set in the antecedent of rule is at least equal to 0.5. In this case, the threshold helps to filter those rules that are no representative for the pattern $\widehat{\mathbf{x}}$. Moreover, in the experiments, we employed the best rule reasoning method by setting the value of threshold to 0. In this case, a rule is considered any time has been activated by the pattern $\widehat{\mathbf{x}}$.

3.5.2 Experimental Study

To characterize the behavior of the proposed methodology for generating fuzzy associative classifiers for big data, the experimental study focuses on two aspects. First, we investigate the performance of DAC-FFP in terms of classification accuracy, model complexity and computation time. Then, we carry out a scalability analysis which includes several tests varying the number of nodes and the size of the datasets.

We tested our algorithm on six well-known big datasets, extracted from the UCI repository[5] and LIBSVM repository[6], As shown in Table 3.16, datasets are characterized by different numbers of input/output instances (up to 11000000), classes (from 2 to 23), and attributes (up to 54). For each dataset, we report the number of numeric (N) and categorical (C) attributes.

All the experiments have been run using a typical low-end system testbed for supporting the target classification service: a small cluster with one master with 4-core CPU (Intel Core i5 CPU 750 x 2.67 GHz), 4 GB of RAM and a 500GB Hard Drive, and four slave nodes with 4-core CPU with Hyperthreading (Intel Core i7-2600K CPU x 3.40 GHz), 16GB of RAM and 1 TB Hard Drive per each. All nodes are connected by a Gigabit Ethernet (1 Gbps) and run Ubuntu 12.04. The algorithm has been deployed upon Apache

[5] Available at `https://archive.ics.uci.edu/ml/datasets.html`
[6] Available at `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/`

Table 3.16: Big datasets used in the experiments.

Dataset	# Instances	# Attributes	# Classes
Cover Type (COV)	581012	54 (10 N, 44 C)	2
HIGGS (HIG)	11000000	28 (28 N)	2
KDDCup 1999 2 Classes (KDD99_2)	4856151	41 (26N, 15C)	2
KDDCup 1999 5 Classes (KDD99_5)	4898431	41 (26N, 15C)	5
KDDCup 1999 (KDD99)	4898431	41 (26N, 15C)	23
Susy (SUS)	5000000	18 (18 N)	2

Hadoop 1.0.4 as MapReduce implementation: the master hosts the *NameNode* and *JobTracker* processes, while each slave runs a *DataNode* and a *TaskTracker*.

Analysis of the fuzzy associative classifier performance

In this section, we analyze the performance of DAC-FFP in terms of accuracy, model complexity, and computation time. In particular, we show the results obtained by DAC-FFP and, with the main aim of showing that the accuracies of the proposed classifier are comparable with the ones achieved by recent state-of-the-art algorithms, we discuss also the results achieved by two crisp associative classifiers, namely MRAC and MRAC+ [17]. For the sake of completeness, we mention that to the best of our knowledge only one other fuzzy classification approach has been proposed for dealing with big datasets [119]. Here, as discussed in the Introduction, the authors present a distributed version of the Chi et al's algorithm [36]. Since the authors adopt a ten-fold cross-validation and a fair comparison is not possible, we have not shown the results achieved by such method in Table 3.18.

As described in Section 3.2.2, MRAC can be viewed as an extension of the well-known CMAR [108] algorithm in a distributed execution environment and consists in three main phases. First, it adopts a MapReduce discretization step of the well-known multi-interval discretization approach proposed by Fayyad and Irani [61] to split the domain of each continuous feature into bins. Items are identified by assigning a categorical value to each bin. Second, the method extracts frequent CARs with support, confidence, and chi-squared higher than pre-fixed thresholds by employing a parallel version of the FP-growth algorithm [106] adapted for classification problems. Third, two rule pruning steps based on redundancy and training set coverage are applied to generate the final rule base. As well as in CMAR approach, MRAC employ the weighted chi-squared as reasoning method. MRAC+, in Section 3.2.3, represents an enhanced version of MRAC with the aim to improve the performance of the algorithm in terms of accuracy and execution time. In particular, MRAC+ avoids the training set coverage pruning step that represent one of the most consuming part of the algorithm and employs the a crisp version of the best rule as inference mechanism. Both algorithms have proved to be very effective comparing with two well-known distributed implementations of classifier, namely Decision Tree and Random Forest, available on MLlib [140] and Mahout [127], respectively. In particular, the experimental study shows that MRAC+ outperforms the decision tree and achieves comparable results with Random Forest in terms of accuracy and execution time.

Table 3.17 summarizes the parameters used in the experiments. We highlight that for MRAC and MRAC+, we have just extracted, for the common datasets, the results from the related paper [17]. However, for sake of clarity, we also report the parameters used by both algorithms. For all algorithms, a five-fold cross-validation, using the same data partitions, has been carried out.

Table 3.17: Values of the parameters for each algorithm used in the experiments.

Method	Parameters
DAC-FFP	$\gamma = 0.1\%, \phi = 2\%, MinSupp = 0.01\%, MinConf = 50\%, min\chi^2 = 20\%, K = 15000, \delta = 4$
MRAC+	$\gamma = 0.1\%, \phi = 2\%, MinSupp = 0.01\%, MinConf = 50\%, min\chi^2 = 20\%, K = 15000$
MRAC	$\gamma = 0.1\%, \phi = 2\%, MinSupp = 0.01\%, MinConf = 50\%, min\chi^2 = 20\%, K = 15000, \delta = 4$

Table 3.18 shows, for each dataset and for each algorithm, the average values $\pm$ standard deviation of the accuracy, both on the training (Acc_{Tr}) and test sets (Acc_{Ts}) obtained by the three algorithms. The highest accuracy values for each dataset are shown in bold. As regards DAC-FFP, we report also the results achieved by each one of the four experimented reasoning methods; here WV, MM, $BFR(0)$ and $BFR(1)$ stand for *weighting vote*, *maximum matching*, *best rule* with no threshold and *best rule* with threshold equal to 1, respectively.

Table 3.18: Average accuracy $\pm$ standard deviation achieved by DAC-FFP, MRAC+ and MRAC.

		DAC-FFP		MRAC+		MRAC	
Dataset	Inference	Acc_{Tr}	Acc_{Ts}	Acc_{Tr}	Acc_{Ts}	Acc_{Tr}	Acc_{Ts}
COV	WV	77.227 ± 0.043	77.190 ± 0.212	$\mathbf{78.329} \pm 0.091$	$\mathbf{78.092} \pm 0.157$	74.246 ± 0.100	74.261 ± 0.156
	MM	75.194 ± 0.045	75.052 ± 0.207				
	BFR (0)	75.550 ± 0.059	75.519 ± 0.170				
	BFR (1)	76.943 ± 0.124	76.877 ± 0.174				
HIG	WV	$\mathbf{66.019} \pm 0.062$	$\mathbf{66.005} \pm 0.078$	65.942 ± 0.058	65.904 ± 0.045	65.079 ± 0.054	65.050 ± 0.061
	MM	63.486 ± 0.066	63.472 ± 0.066				
	BFR (0)	65.514 ± 0.033	65.507 ± 0.022				
	BFR (1)	65.738 ± 0.036	65.733 ± 0.023				
KDD99_2	WV	99.987 ± 0.011	99.986 ± 0.011	$\mathbf{99.999} \pm 0.000$	$\mathbf{99.998} \pm 0.000$	99.959 ± 0.002	99.957 ± 0.004
	MM	99.997 ± 0.000	99.997 ± 0.001				
	BFR (0)	99.998 ± 0.000	$\mathbf{99.998} \pm 0.001$				
	BFR (1)	$\mathbf{99.999} \pm 0.001$	$\mathbf{99.998} \pm 0.001$				
KDD99_5	WV	99.935 ± 0.029	99.935 ± 0.031	99.863 ± 0.046	99.858 ± 0.047	99.898 ± 0.034	99.898 ± 0.035
	MM	99.915 ± 0.054	99.914 ± 0.054				
	BFR (0)	$\mathbf{99.941} \pm 0.031$	$\mathbf{99.939} \pm 0.032$				
	BFR (1)	$\mathbf{99.941} \pm 0.031$	$\mathbf{99.939} \pm 0.032$				
KDD99	WV	99.839 ± 0.123	99.838 ± 0.123	99.582 ± 0.020	99.579 ± 0.020	99.640 ± 0.024	99.639 ± 0.024
	MM	99.845 ± 0.134	99.843 ± 0.135				
	BFR (0)	$\mathbf{99.887} \pm 0.150$	$\mathbf{99.886} \pm 0.150$				
	BFR (1)	$\mathbf{99.887} \pm 0.152$	$\mathbf{99.886} \pm 0.150$				
SUS	WV	77.728 ± 0.022	77.716 ± 0.039	78.247 ± 0.013	78.220 ± 0.035	76.245 ± 0.055	76.232 ± 0.068
	MM	76.670 ± 0.025	76.664 ± 0.037				
	BFR (0)	77.526 ± 0.038	77.528 ± 0.076				
	BFR (1)	$\mathbf{78.274} \pm 0.004$	$\mathbf{78.267} \pm 0.050$				

The analysis of Table 3.18 highlights that for the different versions of the KDD datasets the reasoning method does not influence very much the final results. Indeed, we verified that for these datasets all rules have a confidence value higher than 99.6%: for this reason, all the experimented reasoning methods select more or less the same rules for classifying unlabeled pattern and led to similar classification accuracies. As regards SUS datasets, we notice that the best accuracy is achieved when the BRF(1) reasoning method is employed: this is due to the fact that the generated rule bases contain, on average, 28% and 64.6% of rules with confidence higher than 90% and 80%, respectively. Finally, as regards COV and HIG datasets, we realize that the best accuracies are obtained when the WV reasoning method is used: indeed, in these cases, the number of rules with very high confidence is not so high. Hence, most of the rules are not able to depict a good representation of the training set and a method that takes care of the vote of all rules is more suitable than an approach which select only one rule for classifying an unlabeled pattern. On the basis of the aforementioned results, we can advise to use the BRF reasoning method whenever the final rule base contain a good percentage of rules characterized by a high confidence. On the other hand, when the amount of confident rules is reduced, the weighted vote reasoning method can help to improve the number of patterns correctly classified. Finally, if most of the rules is associated with a very high confidence, each reasoning method can be used for pattern classification.

As regards the accuracy level achieved by DAC-FPP, Table 3.18 highlights that, on average, DAC-FFP outperforms MRAC in all datasets, both on training and test sets. On the other hand, DAC-FFP and MRAC+ achieve similar accuracies on 4 datasets, while on COV dataset MRAC+ performs better than DAC-FPP and on HIG dataset DAC-FPP with WV outperforms MRAC+. In general, we can state that the proposed method for generating fuzzy association rules allow us to achieve similar or better accuracies with respect to two recent algorithms for generating classifiers for big data.

If we analyze the results shown in Table 3.19, we realize that the complexities, expressed in terms of average number of rules ($\#Rules$), of the models generated employing the DAC-FFP are much more lower than the complexities associated to both MRAC+ and MRAC. In particular, we notice that the number of rules of DAC-FFP is almost always one order of magnitude less than MRAC+ and of the same order of magnitude of MRAC. In conclusion, the fuzzy associative classifiers generated by the proposed methodology are characterized by a lower number of parameters, namely the rules, than the ones generated by two recent algorithms which generate crisp rule-based classifiers. On the other hand, even though DAC-FFP employs a lower number of rules than both MRAC and MRAC+, the generated models are not still very interpretable. However, since the activated rules do not contain, in general, a large number of conditions (at most four or five on average for all datasets), we can affirm that a reasoning method such as the best rule that consider just one rule in the inference process, can provide a very interpretable explanation for each conclusion. Thus, even though the number of rules is still quite large in the final rule base, we can consider that, for each unlabeled pattern, the classifier can provide a very intuitive justification of its reasoning.

Table 3.19: Complexities of DAC-FFP, MRAC+ and MRAC.

Dataset	DAC-FFP	MRAC+	MRAC
COV	2,646	15,612	6,714
HIG	9,365	29,999	19,468
KDD99_2	890	30,000	1,174
KDD99_5	2,419	49,349	2,878
KDD99	2,347	125,294	2,806
SUS	10,970	30,000	21,963

Table 3.20 summarizes the computation times (in seconds) spent by each algorithm. All algorithms use the same Hadoop configuration: the number of mappers and reducers is set equal to the available cores on the cluster. It is worth noting that due to memory constraint, for DAC-FFP as well as MRAC+ and MRAC, the number of reducers for the *parallel FP-Growth* is set equal to 4 and, only for HIG dataset, is set equal to 2. Moreover, we also report the number of HDFS blocks (Z) and instances per block (N_{Block}). As shown in Table 3.20, DAC-FFP is slower than the other comparison algorithms. In particular, DAC-FFP employs an additional pruning step than MRAC based on one MapReduce job; increasing the execution time of the overall learning process.

Table 3.20: The computation times (in seconds) for the learning process in DAC-FFP, MRAC+, MRAC.

Dataset	$\mathbf{Z}$	$\mathbf{N_{Block}}$	DAC-FFP	MRAC+	MRAC
COV	1	464,809	1,382	504	1,059
HIG	96	91,667	21,978	6,141	9,881
KDD99_2	6	47,487	9,857	669	704
KDD99_5	6	653,124	23,410	1,439	1,708
KDD99	6	653,124	70,828	1,878	2,280
SUS	29	137,932	10,770	738	3,713

Scalability Analysis

In this section, we investigate the scalability of DAC-FFP, analyzing how the different phases affect the performance of the algorithm. In particular, we carried out several tests varying the number of CUs. Similar to MRAC+ in Section 3.2.4, we use the common measure employed in parallel computing, namely the *speedup* σ. As stated by the speedup definition, the efficiency of a program, which employs multiple CUs, can be calculated comparing the execution time of the parallel implementation against the corresponding sequential version. Unfortunately, because of the large size of the employed datasets, the sequential implementation of the algorithm is impracticable and would take an unreasonable time to be fully executed. To overcome this drawback, we adopt a slightly different definition, taking as reference a run over Q identical cores, with $Q > 1$. In particular, we

redefine the speedup formula on n CUs as $\sigma_{Q^*}(n) = Q^* \cdot \tau(Q^*)/\tau(n)$, where $\tau(n)$ is the algorithm runtime using n CUs and Q^* is the number of CUs used to run the reference execution. Obviously, $Q^*(n)$ makes sense only for $n \geq Q^*$, where the speedup is expected to be sub-linear due to the overhead from the Hadoop procedures, the contention of shared resources among cores, and the contention of network bandwidth among machines. In our experiments, we have set $Q^* = 8$ and performed several executions, varying the number of CUs, with $n > 8$. To avoid unbalanced loads, we have kept the same number of running cores per node, adding the number of switched-on slave nodes. Practically, we have recorded the execution times of each experiment, starting from 1 to 4 slave nodes (from 8 to 32 cores).

Table 3.21 summarizes the results. Figures 3.24*a* and 3.24*b* show the execution time and the runtime, respectively, obtained on the whole Susy dataset. Similar results can be recorded with the other datasets. We recall that with the default Hadoop settings, the number Z of mappers is automatically determined by the HDFS block size. Obviously, only Q mappers can be run simultaneously and the rest ($Z - Q$) are queued, waiting for being scheduled and running whenever one of the running mappers completes. Thus, in the ideal case of the same execution time for all the mappers, the map phase for each MapReduce stage would require $\lceil \frac{Z}{Q} \rceil$ iterations. In case $Z \leq Q$ then the global runtime is practically driven by the longest of the mappers' runtime. As regards Susy dataset, Hadoop instantiates 36 mappers; thus the number of iterations corresponds to 5, 3, 2 and 2 iterations on 8, 16, 24 and 32 cores, respectively. Same considerations can be made for the reduce phase too. In this case, the number R of reducers is defined by the user and in our experiments, considering the structure of our algorithm we have set R equal to the number of available cores, with the only exception for the *Parallel Fuzzy FP-Growth* phase where, due to memory constraints, we have limited R to 4 per machine. Each reducer processes sequentially a set of keys generated during the map phase and each key is assigned to only one reducer according the default partitioning function $hash(key)\ mod\ R$.

Table 3.21: Speedup of the overall algorithm for the Susy dataset.

# Cores	**Time (s)**	**Speedup**	$\sigma_8(Q)/Q$ (Utilization)
8	50,740	8	1.000
16	27,251	14.896	0.931
24	19,678	20.628	0.860
32	15,482	26.219	0.819

As shown in Figures 3.24*a* and 3.24*b*, the actual speedup σ_8 and the utilization index tend to decrease quite rapidly, due to the increasing overhead of the Hadoop procedures (adding a new slave node implies adding a new *Datanode* and *Tasktracker* as well) and of the networks communications between master and slaves. However, within the limitations

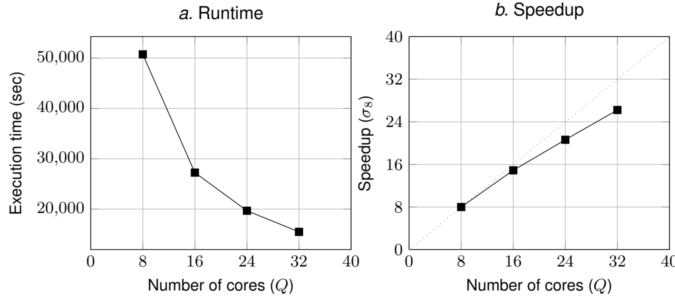


Figure 3.24: Execution time and speedup of DAC-FFP on SUS dataset for increasing numbers of cores

due to the different experimental settings, this result is in line with [106] and MRAC+ [17], where the value of the utilization index are 0.768 and 0.80, respectively.

However, a better understanding of the presented overall figures requires a deeper insight of the contributions from the different MapReduce jobs. To this aim, in Table 3.22 and Figure 3.25 we report the speedup σ_8 of the fuzzy partitioning and the four most time consuming jobs, namely *Distributed Fuzzy Partitioning*, *Parallel Fuzzy FP-Growth*, *Pruning based on Fuzzy Supp and Conf*, and *Pruning based on Fuzzy Training Set Coverage*. The contribution of the other two phases, namely *Parallel Fuzzy Counting* and *Parallel Rules Selection*, is negligible.

Table 3.22: Runtime and speedup of each DAC-FFP phase in the Susy dataset.

	Distributed Fuzzy Partitioning			Parallel Fuzzy FP-Growth			Pruning based on Fuzzy Supp and Conf			Pruning based on Fuzzy Training Set Coverage		
# Cores	Time (s)	Speedup σ_8	$\sigma_8(Q)/Q$	Time (s)	Speedup σ_8	$\sigma_8(Q)/Q$	Time (s)	Speedup σ_8	$\sigma_8(Q)/Q$	Time (s)	Speedup σ_8	$\sigma_8(Q)/Q$
8	787	8	1	3,161	8	1	40,790	8	1	5,557	8	1
16	625	10.074	0.630	1,688	14.981	0.936	21,612	15.099	0.944	2,929	15.178	0.949
24	504	12.492	0.521	1,249	20.247	0.844	15,556	20.977	0.874	2,104	21.129	0.880
32	484	13.008	0.407	987	25.621	0.801	12,137	26.886	0.840	1,641	27.091	0.847

The four charts in Figure 3.25 clearly show that the phases behave differently with respect to scalability. As shown in Figure 3.25*a*, the speedup of *Distributed Fuzzy Partitioning* rapidly decreases and using 32 cores does not produce a significant advantage; indeed the execution time from 24 to 32 cores drops down of only few seconds. The result is mainly affected by the runtime of the last reduce step devoted to the generation of strong fuzzy partitions. Such step represents the most time consuming part of the *Distributed Fuzzy Partitioning*. Since Susy is characterized by 18 continuous attributes, each reducer processes approximately 3, 2, 1 and 1 attributes in case of 8, 16, 24 and 32 cores, respectively. The result highlights that, as regards the distribution of the computational flow of the reduce phase, using a number of cores higher than 18 does not produce a real advantage. On the other hand, the global execution time of the map phases can be shrunk by reducing the number of iterations, i.e. by exploiting a proper number of cores.

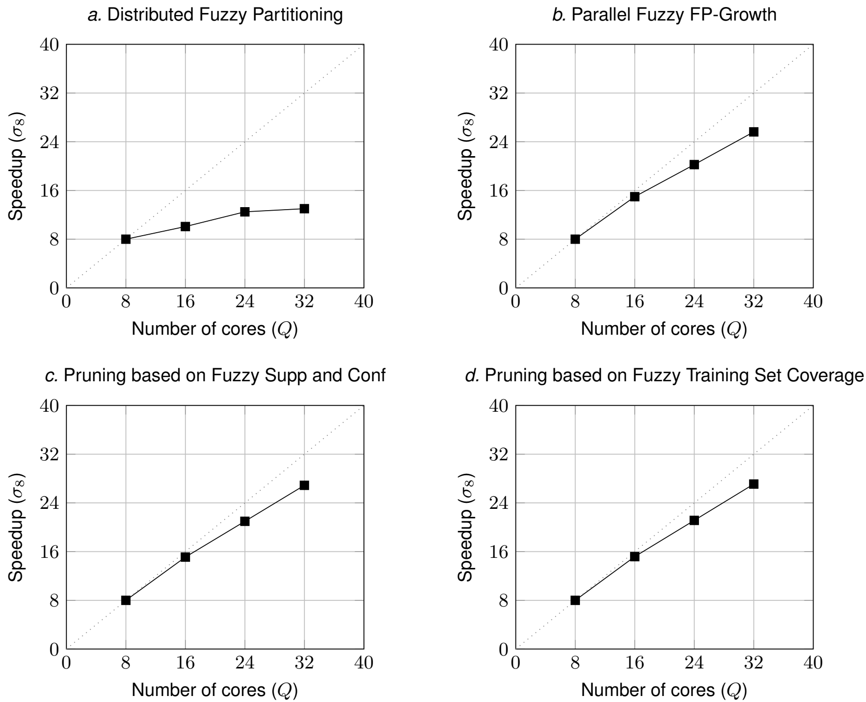


Figure 3.25: Speedup of the different phases of DAC-FFP on SUS dataset for increasing number of cores

Like the *Distributed Fuzzy Partitioning* phase, the computational weight of the reducing activity negatively affects the speedup of the *Parallel Fuzzy FP-Growth* phase (Figure 3.25*b*). Indeed, the average runtime of each mapper is quite short (a few seconds), and the global execution time is dominated by the mining of FCARs from each conditional FP-Tree. With the Hadoop default settings, all the conditional FP-Trees are evenly distributed among the reducers and adding more cores helps us to improve the FP-Growth parallelization, decreasing the number of conditional FP-Trees processed by each reducer. We recall that due to memory constraints, for the *Parallel Fuzzy FP-Growth* phase we set the number of reducers to 4 per machine. Hence, since in our tests Susy has 290 frequent items, each reducer processes more or less 73, 37, 25 and 19 FP-Trees in case of 4, 8, 12 and 16 reducers, respectively. We highlight that such result shows a very similar trend to the Parallel FP-Growth phase of both the MRAC+ and MRAC [17].

On the other hand, both *Pruning based on Fuzzy Supp and Conf* and *Pruning based on Fuzzy Training Set Coverage* (Figures 3.25*c* and 3.25*d*, respectively) present a similar behaviour, with satisfactory utilization values. In this case, the overall runtime is mainly driven by the map phase and the global execution time can be shrunk by reducing the number of iterations, i.e. by exploiting additional cores, as witnessed by the results in Table 3.22.

4

Tree based Classification

Decision trees are very popular classification methods successfully employed in many application domains. Fuzzy decision trees (FDTs) are an extension of crisp decision trees to deal with uncertain data. However, FDTs are characterized by higher complexity than crisp decision trees. First, most of the FDTs require that a fuzzy partition has been already defined upon each continuous attribute. Second, the splitting method used in generating child nodes from a parent node generally optimizes purposely-defined index, exploiting particular fuzzy partitions or quite complex approaches. Third, each node in fuzzy decision trees represents a fuzzy subset rather than a crisp set as in classical crisp decision trees. Thus, each instance can activate different branches and reach multiple leaves, increasing the complexity of the overall learning process. Moreover, FDTs have been mainly used in the literature for classifying small datasets and they have focused on increasing classification accuracy, often neglecting time and space requirements.

In this chapter, we propose a distributed fuzzy discretizer and a distributed FDT (DFDT) learning scheme upon the MapReduce programming model for managing big data. The discretizer generates a Ruspini fuzzy partition for each continuous attribute by using a purposely adapted distributed version of the well-known method proposed by Fayyad and Irani in [61]. The fuzzy partitions computed by the discretizer are used as input to the DFDT learning algorithm. We adopt and compare two different versions of the learning algorithm based on binary and multi-way splits, respectively. Both the versions employ the information gain computed in terms of fuzzy entropy for selecting the attribute to be adopted at each decision node.

We have implemented both the discretizer and the learning scheme on Apache Spark. We have used eight real-world big datasets characterized by a different number of instances (up to 11 millions) and class labels (from 2 to 23). We have compared the results obtained by our approach with the ones achieved by a state-of-the-art distributed decision tree learning algorithm implemented in the MLLib on Spark with respect to accuracy, complexity and scalability.

The chapter is organized as follows. Section 4.1 describes fuzzy decision trees, discussing some related works in the framework of distributed discretization algorithms and

distributed decision trees, and introducing the necessary notations used in the rest of the chapter (Section 4.1.1). Section 4.2 first introduces the fuzzy discretizer and the FDT learning algorithm and then discusses their distributed implementation, detailing each single MapReduce job. Section 4.3 presents and discusses the experimental results comparing the proposed approach with one state-of-the-art distributed classifier in terms of accuracy, complexity and computational time.

4.1 Fuzzy Decision Trees

Decision trees are widely used classifiers, successfully employed in many application domains such as security assessment [50, 193], health system [42, 72, 142] and road traffic congestion [210]. The popularity of the decision trees is mainly due to the simplicity of their learning schema, which however allows achieving accuracies comparable to other well-known classification approaches. Further, decision trees are considered among the most interpretable classifiers [79, 162, 188, 197], that is, they can explain how an output is inferred from the inputs. Finally, the tree learning process usually requires only a few parameters that must be set.

Several algorithms have been proposed in the last decades for generating decision trees: most of them are extensions or improvements of the well-known ID3 proposed by Quinlan et al. [154] and CART proposed by Brieman et al. [24]. In a decision tree, each internal (non-leaf) node denotes a test on an attribute, each branch represents the outcome of the test, and each leaf (or terminal) node holds a class label.

Several works have exploited the possibility of integrating decision trees with the fuzzy set theory to deal with uncertainty [33, 93], leading to the so-called fuzzy decision trees (FDTs). Unlike crisp decision trees, each node in FDTs is characterized by a fuzzy set rather than a crisp set. Thus, Moreover, Both crisp and fuzzy decision trees are generated by applying a top-down approach that partitions the training data into homogeneous subsets, that is, subsets of instances belonging to the same class [155]

Like crisp decision trees, FDTs can be categorized into two main groups, depending on the splitting method used in generating child nodes from a parent node [116]: *binary* (or two-way) split trees and *multi-way* split trees. Binary split trees are characterized by recursively partitioning the attribute space into two subspaces so that each parent node is connected exactly with two child nodes. On the other hand, multi-way split trees partition the space into a number of subspaces so that each parent node generates in general more than two child nodes. Since a tree with multi-way splits can be always redrawn as a binary tree [81], apparently the use of multi-way split seems to offer no advantage. We have to consider, however, that binary split implies that an attribute can be used several times in the same path from the root to a leaf. Thus, the binary split tree is characterized by a higher number of leaves, is deeper, and sometimes harder to interpret for experts than multi-way splits [18, 99]. Further, in some domain [99], multi-way splits seem to lead to more accurate trees but, since multi-way splits tend to fragment the training data very quickly [81], they generally need larger data size in order to work effectively.

Generally, FDT learning algorithms require that a fuzzy partition has been already defined upon each continuous attribute. For this reason, each continuous attribute is usually discretized by optimizing purposely-defined indexes [191, 203]. Discretization can drastically affect the accuracy of classifiers [68, 100, 208] and therefore should be performed with great care. In [208], authors perform an interesting analysis by investigating how different discretization approaches influence the accuracy and the complexity (in terms of number of nodes) of the generated fuzzy trees: they employ several well-known fuzzy partitioning methods and different approaches for, given a crisp partition generated by well-known discretization algorithms [68, 100], defining membership functions such as triangular, trapezoidal and Gaussian. The experimental results on 111 different combinations highlight that seven of them outperform the others both in accuracy and number of nodes.

FDTs have been mainly used in the literature for classifying small datasets. Thus, FDT learning approaches have focused on increasing classification accuracy, often neglecting time and space requirements, by adopting several heavy tasks such as pruning steps, genetic algorithms, and computation of the optimal split among all points at each node [28, 44, 92, 144]. Thus, these approaches are not generally suitable for dealing with a huge amount of data. A possible simple solution for applying these approaches would be to select only a subset of data objects by applying some downsampling technique. However, these techniques may delete useful knowledge, making FDT learning approaches purposely designed for managing the overall dataset more desirable and effective. In our context, this means explicitly addressing Big Data.

In the last years, some decision tree learning algorithms have been proposed for managing big data by adopting the MapReduce paradigm on the top of Apache Hadoop [43, 186, 187]. MapReduce is based on functional programming and divides the computational flow into two main phases, namely Map and Reduce, which communicate by $\langle key, value \rangle$ pairs. The MapReduce implementation of a distributed decision tree proposed in [43] employs, for instance, four map-reduce stages. The first stage scans the dataset for creating the initial data structures employed in the other three stages. These stages are executed iteratively for, respectively, (i) selecting the best attribute, (ii) updating the statistics for the new nodes and (iii) growing the tree. The experimental results discussed in the paper are limited only to the scalability analysis by varying the number of nodes and the dataset size (up to 3 millions of instances). The effectiveness of the decision trees for managing big data has been proved in real application domains such as stock futures prediction [184] and clinical decision support [129]. Other works [48, 105, 176] exploit decision trees for generating ensemble of classifiers such as random forest. To deal with big data, the proposed algorithms first build concurrently multiple trees from different chunks of data and then group all of them for generating the forest. However, the generation of each tree is not distributed on the cluster but is performed sequentially on a single chunk of the entire dataset.

However, as introduced in Secition 1, only a few classifiers proposed for managing big data employ fuzzy sets. In [119], the authors describe Chi-FRBCS-BigData, a fuzzy

rule-based classification system based on the Chi et al.'s approach [36]. This approach has been modified to deal with big data by employing two map-reduce stages. The first stage builds the model from chunks of the training set: a group of fuzzy rules is generated from each chunk. Then, these groups are fused together in the reduce phase. The second stage estimates the class using the model learned in the first stage. The authors have investigated different approaches for fusing the fuzzy rules by developing two different versions, named Chi-FRBCS-BigData-Max and Chi-FRBCS-BigData-Ave. Moreover, an improved version, called Chi-FRBCS-BigDataCS, has been proposed in [120] for handling imbalanced big datasets. As regards FDTs, in [58], authors have proposed FDT 2.0, an improved version of FDT, obtained by integrating the FDT approach into the modern database technology for improving the scalability of the overall algorithm and handling large data sets. However, the tests have been carried out on datasets, which involve at most 400,000 instances, using MySQL as modern database manager. Thus, FDT 2.0 cannot be considered a solution for managing big datasets.

Most of the classical FDT implementations proposed in the literature require that a fuzzy partition is defined on each continuous attribute before starting the tree learning: the partition is generally obtained by adopting heuristic approaches, which optimize purposely-defined indexes [190, 191, 203]. Actually, the discretization process is crucial to performance of FDTs, especially when a huge amount of data is involved. Recently, in [157, 158] a distributed implementation[1] of the well-known Entropy Minimization Discretizer [61] has been proposed in order to partition continuous attributes. The algorithm first distributes the computation of the class frequency for each attribute, then sorts the values of each attribute in ascending order and selects a set of interval boundaries by retrieving those values that fall in the class borders. Finally, the generation of cut-points based on the entropy information is performed. For all attributes characterized by a low number of boundary values (lower than a fixed threshold), the computation can be processed independently in a single step. On the other hand, for attributes characterized by a high number of boundary values, the selection of the best cut-points has to be carried out iteratively. The first case is obviously more efficient. The second case, although less efficient, happens more rarely. The algorithm has been tested by using two datasets (up to about 65 millions of instances) and the generated cut points have been employed by the distributed Naive Bayes classifier available on the MLlib Library.

Although in the last decades different methods [36, 134] have been investigated for generating suitable membership functions (MFs) for each linguistic variable, only a few works have addressed this task for big data. For instance, in [119] and [120], authors use the method proposed by Chi et al. [36], that is an extension of the well-known Wang and Mendel algorithm [185], for tackling classification problems. However, this approach requires that the fuzzy partitions are already available; otherwise they can be generated by equally distributing a fixed number of triangular fuzzy sets on the domain of each attribute. A similar approach is performed in [167], where authors introduce a novel model

[1] The algorithm has been integrated as a third-party package of MLlib Library and can be downloaded from `https://github.com/sramirez/spark-MDLP-discretization`

for time series forecasting based on the hybridization of fuzzy sets and artificial neural networks. However, due to involvement of voluminous data, the time series is fuzzified by adopting a two-step algorithm: first the domain of each attribute is partitioned by using a fixed number of equal length intervals and then a fuzzy partition is defined on these intervals by employing triangular fuzzy sets. Even if the complexity of the proposed approaches is independent of the number of data, the equal-width binning is sensitive to outliers and many data points could fall into one or a few bins. Thus the generated fuzzy partitions could be not meaningful for the distribution of data, affecting the performance of the models [208].

4.1.1 Background

Instance classification consists of assigning a class C_m from a predefined set $C = \{C_1, \ldots, C_M\}$ of M classes to an unlabeled instance. Each instance can be described by both numerical and categorical attributes. Let $\mathbf{X} = \{X_1, \ldots, X_F\}$ be the set of attributes. In case of numerical attributes, X_f is defined on a universe $U_f \subset \Re$. In case of categorical attributes, X_f is defined on a set $L_f = \{L_{f,1}, \ldots, L_{f,T_f}\}$ of categorical values.

An FDT is a directed acyclic graph, where each internal (non-leaf) node denotes a test on an attribute, each branch represents the outcome of the test, and each leaf (or terminal) node holds one or more class labels. The topmost node is the *root* node. In general, each leaf node is labeled with one or more classes C_m with an associated weight w_m: weight w_m determines the strength of class C_m in the leaf node.

Let $TR = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), ..., (\mathbf{x}_N, y_N)\}$ be the training set, where, for each instance $(\mathbf{x}_i, y_i)$, with $i = 1, ..., N$, $y_i \in C$ and $x_{i,f} \in U_f$ in case of continuous attribute and $x_{i,f} \in L_f$ in case of categorical attribute, with $f = 1, ..., F$. FDTs are generated in a top-down way by performing recursive partitions of the attribute space. Algorithm 1 shows the scheme of a generic FDT learning process.

The $SelectAttribute$ procedure selects the attribute used in the decision node and determines the splits generated from the values of this attribute. The selection of the attribute is performed by using appropriate metrics, which measure the difference between the levels of homogeneity of the class labels in the parent node and in the child nodes generated by the splits. The most popular of these metrics are the fuzzy information gain [208], fuzzy Gini index [28], minimal ambiguity of a possibility distribution [203], maximum classification importance of attribute contributing to its consequent [189] and normalized fuzzy Kolmogorov-Smirnov discrimination quality measure [22]. In this work, we adopt the fuzzy information gain, which will be defined in Section 4.2.2. The splitting method adopted in the $SelectAttribute$ procedure determines the attribute to be selected and also the number of child nodes. In the literature, both multi-way and binary splits are used. We have implemented both the approaches and evaluated pros and cons of them.

After the tree has been generated, a given unlabeled instance $\widehat{\mathbf{x}}$ is assigned to a class $C_m \in C$ by following the activation of nodes from the root to one or more leaves. In classical crisp decision trees, each node represents a crisp set and each leaf is labeled

Algorithm 1 Pseudo code of a generic FDT learning process.

Require: training set TR, set X of attributes, splitting method $SplitMet$, stopping method $StopMet$

```
 1: procedure FDTLEARNING(in: TR, X, SplitMet, StopMet)
 2:     root ← create a new node
 3:     tree ← TREEGROWING(root, TR, X, SplitMet, StopMet)
 4:     return tree
 5: end procedure
 6: procedure TREEGROWING(in: node, S, X, SplitMet, StopMet)
 7:     if STOPMET(node) then
 8:         node ← mark node as leaf
 9:     else
10:         splits ← SELECTATTRIBUTE(X, S, SplitMet)
11:         for each split_z in splits do
12:             S_z ← get the set of instances from S determined by split_z
13:             child_z ← create one node by using split_z and S_z
14:             node ← connect the node with TREEGROWING(child_z, S_z, X_z, SplitMet,
    StopMet)
15:         end for
16:     end if
17:     return node
18: end procedure
```

with a unique class label. It follows that $\widehat{\mathbf{x}}$ activates a unique path and is assigned to a unique class. In FDT, each node represents a fuzzy subset. Thus, $\widehat{\mathbf{x}}$ can activate multiple paths in the tree, reaching more than one leaf with different strengths of activation, named *matching degrees*. Given a current node CN, the matching degree $md^{CN}(\widehat{\mathbf{x}})$ of $\widehat{\mathbf{x}}$ with CN is calculated as:

$$md^{CN}(\widehat{\mathbf{x}}) = TN(\mu_{CN}(\widehat{x}_f), md^{PN}(\widehat{\mathbf{x}})) \tag{4.1}$$

where TN is a *t-norm*, $\mu^{CN}(\widehat{x}_f)$ is the membership degree of $\widehat{x}_f$ to the current node CN and $md^{PN}(\widehat{\mathbf{x}})$ is the matching degree of $\widehat{x}_f$ with the parent node PN.

The *association degree* $AD_m^{LN}(\widehat{\mathbf{x}})$ of $\widehat{\mathbf{x}}$ with the class C_m at leaf node LN is calculated as:

$$AD_m^{LN}(\widehat{\mathbf{x}}) = md^{LN}(\widehat{\mathbf{x}}) \cdot w_m^{LN} \tag{4.2}$$

where $md^{LN}(\widehat{\mathbf{x}})$ is the matching degree of $\widehat{\mathbf{x}}$ with LN and w_m^{LN} is the class weight associated with C_m at leaf node LF. In the literature, different definitions have been proposed for weight w_m^{LN} [89, 91]. Further, it has been proved that the use of class weights can increase the performance of fuzzy classifiers.

To determine the output class label of the unlabeled instance $\widehat{\mathbf{x}}$, two different approaches are often adopted in the literature:

- *maximum matching*: the class corresponds to the maximum association degree calculated for the instance;

- *weighed vote*: the class corresponds to the maximum total strength of vote. The total strength of vote for each class is computed by summing all the activation degrees in each leaf for the class. If no leaf has been reached, the instance $\hat{\mathbf{x}}$ is classified as unknown.

  In our approaches, we adopt the weighed vote.

4.2 The Proposed Algorithms

In this section, we introduce the DFDT learning algorithm for handling Big Data. We aim to propose an approach that is easy to implement, is computationally light and guarantees to achieve accuracy values and execution times comparable with other distributed classifiers. We discuss two distinct versions, which differ from each other on the nature of the splitting mechanism.

The workflow of the DFDT learning process consists of two main steps:

1. *Fuzzy Partitioning*: a Ruspini fuzzy partition is determined on each continuous attribute by using a novel discretizer based on fuzzy entropy;
2. *FDT Learning*: an FDT is induced from data by using either a multi-way or a binary splitting mechanism based on the concept of fuzzy information gain.

In the following, we first discuss the two steps in detail and then we describe the adopted distributed implementation for handling Big Data, by specifying how the execution can be parallelized and distributed among the Computing Units (CUs) available on the cluster. We highlight that our approach can be easily employed to perform different tree learning procedures with different membership functions in any cloud-computing environment.

4.2.1 Fuzzy Partitioning

As discussed in Section 4.1, partitioning of the continuous attributes is a crucial aspect in the generation of FDTs. Thus, the choice of how partitioning the continuous variables before executing the decision tree learning algorithm should be performed carefully. An interesting study proposed in [208] has investigated 111 different approaches for generating fuzzy partitions and has analysed how these approaches can influence the accuracy and the complexity (in terms of number of nodes) of the generated FDTs. Among them, Fuzzy Partitioning based on Fuzzy Entropy (FPFE) has proved to be very effective. In this section, we propose an FPFE for generating strong triangular fuzzy partitions when handling big datasets.

The proposed FPFE is an iterative supervised method, which generates *candidate fuzzy partitions* and evaluates these partitions employing the fuzzy information gain, computed using the fuzzy entropy. The algorithm selects the candidate fuzzy partition that maximizes the fuzzy information gain and then splits the continuous attribute domain into two subsets. Similar to the Entropy Minimization method proposed by Fayyad and Irani

in [61], the process is repeated for each generated subset until the Minimum Description Length Principle (MDLP) stopping criterion is satisfied. The candidate fuzzy partitions are generated for each attribute by first sorting all the values of the attribute and determining equi-frequency bins. Then, the bin-boundaries are used to generate candidate fuzzy partition. Since the sorting process is computationally heavy when dealing with big data, we will discuss in Section 4.2.3 an approximated version of the fuzzy partitioning approach, which concurrently performs the sorting operation and the generation of equi-frequency bins on chunks of the training set and then aggregates the lists of bin boundaries for determining a unique sorted list used to generate the candidate fuzzy partitions.

Let $TR_f^0 = [x_{1,f}, ..., x_{N,f}]^T$ be the projection of the training set TR along variable X_f. First of all, the values $x_{i,f}$ are sorted in increasing order. We denote the set of sorted values as S_f. Let l_f and u_f be the lower and upper bounds of S_f. Then, we generate L equi-frequency bins on S_f. For each bin boundary $b_{f,l}$ between l_f and u_f (at the beginning of the partitioning procedure, $l = 1, \ldots, L-1$), we define a Ruspini fuzzy partition of the universe $[u_f, l_f]$ by using three triangular fuzzy sets, namely $B_{f,1}^0$, $B_{f,2}^0$ and $B_{f,3}^0$, as shown in Fig. 4.1. The cores of $B_{f,1}^0$, $B_{f,2}^0$ and $B_{f,3}^0$ coincide with l_f, $b_{f,l}$ and u_f, respectively. Let $S_{f,1}$, $S_{f,2}$ and $S_{f,3}$ be the subsets of points in S_f, which correspond to the supports of $B_{f,1}^0$, $B_{f,2}^0$ and $B_{f,3}^0$, respectively. For each partition induced by $b_{f,l}$, we compute the fuzzy information gain $FGain(b_{f,l}; S_f)$ as:

$$FGain(b_{f,l}; S_f) = FEnt(S_f) - WFEnt(b_{f,l}; S_f) \tag{4.3}$$

where $FEnt(S_f)$ is the fuzzy entropy of S_f and $WFEnt(b_{f,l}; S_f)$ is the weighted fuzzy entropy of the partition induced by $b_{f,l}$. $WFEnt(b_{f,l}; S_f)$ is computed as:

$$WFEnt(b_{f,l}; S_f) = \sum_{j=1}^{3} \frac{|S_{f,j}|}{|S_f|} FEnt(S_{f,j}) \tag{4.4}$$

where $|S_{f,j}|$ and $|S_f|$ are the fuzzy cardinalities of subsets $S_{f,j}$ and S_f, respectively, and $FEnt(S_{f,j})$ is the fuzzy entropy of $S_{f,j}$.

We recall that the fuzzy cardinality of a subset $S_{f,j}$, with membership function $B_{f,j}$ defined on it, is computed as

$$|S_{f,j}| = \sum_{i=1}^{N_{f,j}} \mu_{S_{f,j}}(\mathbf{x}_i) = \sum_{i=1}^{N_{f,j}} \mu_{B_{f,j}}(x_{f,i}) \tag{4.5}$$

where $N_{f,j}$ is the number of points (crisp cardinality) in $S_{f,j}$, $\mu_{S_{f,j}}(\mathbf{x}_i) = \mu_{B_{f,j}}(x_{f,i})$ is the membership degree of $\mathbf{x}_i$ to subset $S_{f,j}$ and $\mu_{B_{f,j}}(x_{f,i})$ is the membership degree of $\mathbf{x}_i$ to fuzzy set $B_{f,j}$. The fuzzy entropy of $S_{f,j}$ is defined as

$$FEnt(S_{f,j}) = \sum_{m=1}^{M} -\frac{|S_{f,j,C_m}|}{|S_{f,j}|} \, log_2(\frac{|S_{f,j,C_m}|}{|S_{f,j}|}) \tag{4.6}$$

where S_{f,j,C_m} is the set of examples in $S_{f,j}$ with class label equal to C_m.

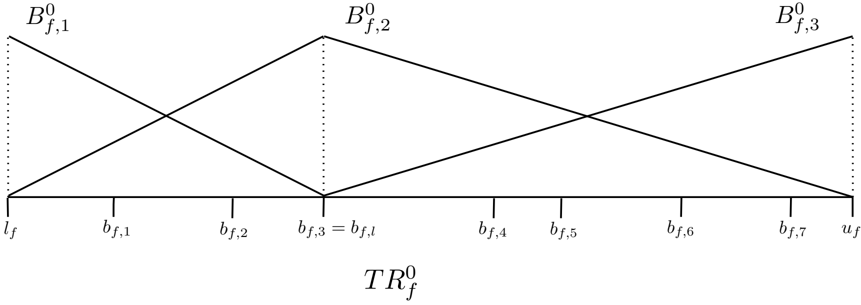


Figure 4.1: An example of fuzzy partition defined on the third bin boundary $b_{f,3}$. We suppose that the domain $[l_f, u_f]$ of S_f has been split into eight equi-freqnecy bins identified by seven bin boundaries $\{b_{f,1}, ..., b_{f,7}\}$

The optimal bin boundary $b^0_{f,l_{max}}$, which maximizes $FGain(b_{f,l}; S_f)$ over all possible candidate fuzzy partitions is selected. The partition is applied if and only if the following condition based on the Minimal Description Length Principle is satisfied:

$$FGain(b^0_{f,l_{max}}; S_f) > \frac{\log_2(|S_f|-1)}{|S_f|} + \frac{\Delta(b^0_{f,l_{max}}; S_f)}{|S_f|} \tag{4.7}$$

where

$$\Delta(b^0_{f,l_{max}}; S_f) = \log_2(3^{k_f} - 2) - \left[k_f \cdot FEnt(S_f) - \sum_{j=1}^{3} k_{f,j} \cdot FEnt(S_{f,j})\right] \tag{4.8}$$

and k_f and $k_{f,j}$ are the numbers of class labels represented in the sets S_f and $S_{f,j}$, respectively.

Let TR^1_f and TR^2_f be the subsets of points of the set TR^0_f, which lie, respectively, in the two intervals $[l_f, b^0_{f,l_{max}}]$ and $(b^0_{f,l_{max}}, u_f]$ identified by $b^0_{f,l_{max}}$. Then, we apply recursively the procedure described above for TR^1_f and TR^2_f by considering $S_f = TR^1_f$ and $S_f = TR^2_f$. The procedure stops when the stopping criterion is met.

Fig. 4.2 shows an example of application of the recursive procedure to the fuzzy partition shown in Fig. 4.1. We can observe that both the partitioning of TR^1_f and TR^2_f are performed, thus generating three fuzzy sets in both $[l_f, b^0_{f,l_{max}}]$ and in $(b^0_{f,l_{max}}, u_f]$. Actually, the two fuzzy sets, which have the core in $b^0_{f,l_{max}}$, are fused for generating a unique fuzzy set. Thus, the resulting partition is a Ruspini partition with five fuzzy sets. This fusion can be applied at each level of the recursion. The final result is a Ruspini fuzzy partition $P_f = [A_{f,1}, ..., A_{f,T_f}]$ on U_f, where $A_{f,j}$ with $j = 1, ..., T_f$ is the jth triangular fuzzy set.

If no partition is possible for attribute X_f, that is, the condition in (4.7) is not satisfied already for TR^0_f, then X_f is discarded and not employed in the FDT learning.

Since triangular MFs are defined by three parameters (a, b, c), where b represents the core and a and c correspond to the lower and upper bounds of the support, respectively, $A_{f,j}$ and $A_{f,j+1}$ are defined as $(a_{f,j}, b_{f,j}, c_{f,j})$ and $(a_{f,j+1} = b_{f,j}, b_{f,j+1} = c_{f,j}, c_{f,j+1})$,

respectively. The fuzzy sets $A_{f,1}$ and A_{f,T_f} defined on P_f are identified by the parameters $(-\infty, l_f, c_{f,1})$ and $(a_{f,T_f}, u_f, +\infty)$, respectively.

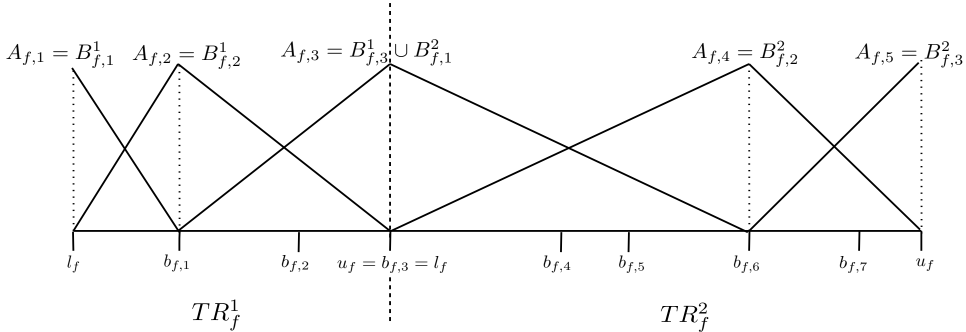


Figure 4.2: An example of application of the recursive procedure to the fuzzy partition shown in Fig. 4.1 ($b^0_{f,l_{max}} = b_3$).

The procedure adopted for the fuzzy partition generation is simple and computationally light. Further, it generates Ruspini fuzzy partitions that are widely assumed to have a high semantic interpretability [67]. Finally, it allows performing an attribute selection because it may lead to the elimination of attributes, speeding up the FDT learning.

4.2.2 Fuzzy Decision Tree Learning

In this section, we introduce the FDT learning algorithm proposed in this paper. We describe two distinct approaches, which differ from each other for the splitting mechanism (binary or multi-way) used in the decision nodes.

Let $P_f = \{A_{f,1}, \ldots, A_{f,T_f}\}$ be a fuzzy partition of T_f fuzzy sets defined on each continuous attribute X_f.

We adopt the FDT learning scheme described in Alg. 1. The $SelectAttribute$ procedure selects the attribute, which maximizes the fuzzy information gain. Then Z child nodes are created. The number of child nodes as well as the computation of the fuzzy information gain depend on the employed splitting method. We have experimented two different methods: binary and multi-way split. The two methods generate Fuzzy Binary Decision Trees (FBDTs) and Fuzzy Multi-way Decision Trees (FMDTs), respectively. Both the trees use fuzzy linguistic terms to specify recursively branching condition of nodes until one of the following termination conditions ($StopMethod$ in Fig. 1) is met:

1. the node contains only instances which belong to the same class;
2. the node contains a number of instances lower than a fixed threshold λ;
3. the tree has reached a maximum fixed depth β;

4. the value of the fuzzy information gain is lower than a fixed threshold ϵ. In our experiments, we set $\epsilon = 10^{-6}$;

In case of multi-way splitting, for each parent node, FMDT generates as many child nodes as the number T_f of linguistic values defined on the splitting attribute X_f: each child node contains only the instances that belong to the support of the fuzzy set corresponding to the linguistic value. Let G be the set of instances in the parent node and G_j be the set of instances in the jth child node. Set G_j contains the instances that belong to the support of $A_{f,j}$. As defined in Eq. 4.5, the cardinality of G_j is defined as:

$$|G_j| = \sum_{i=1}^{N_j} \mu_{G_j}(\mathbf{x}_i) = \sum_{i=1}^{N_j} TN(\mu_{A_{f,j}}(x_{f,i}), \mu_G(\mathbf{x}_i)) \tag{4.9}$$

where N_j is the number of instances (crisp cardinality) in the set G_j, $\mu_{G_j}(\mathbf{x}_i)$ is the membership degree of instance $\mathbf{x}_i$ to set G_j, $\mu_{A_{f,j}}(x_{f,i})$ is the membership degree of instance $\mathbf{x}_i$ to fuzzy set $A_{f,j}$, $\mu_G(\mathbf{x}_i)$ is the membership degree of example $\mathbf{x}_i$ to set G (for the root of the decision tree, $\mu_G(\mathbf{x}_i) = 1$) and the operator TN is a T-norm.

Figure 4.3 illustrates an example of how multi-way splitting is performed. Let us suppose that a fuzzy partition P_f with five triangular fuzzy sets has been defined on a continuous attribute X_f. For a given parent node, the method generates exactly five child nodes, one for each fuzzy set. Let us suppose that, a given instance, represented as a blue circle in Figure 4.3, belongs to $A_{f,1}$ and $A_{f,2}$ with membership degree equal to 0.3 and 0.7, respectively. Thus, the instance belongs to only the child nodes corresponding to $A_{f,1}$ and $A_{f,2}$ and contributes to $|G_1|$ and $|G_2|$ with $TN(0.3, \mu_G(\mathbf{x}_i))$ and $TN(0.7, \mu_G(\mathbf{x}_i))$, respectively.

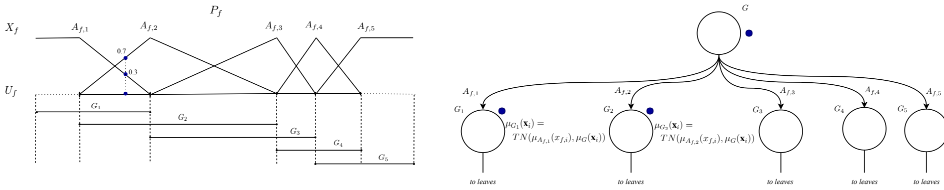


Figure 4.3: An example of multiple splitting on a continuous attribute with five triangular fuzzy sets. The blue circle shows an example of how a given example contributes to the cardinality computation.

The fuzzy information gain $FGain$ for a generic attribute X_f is computed as:

$$FGain(X_f; G_j) = FEnt(G_j) - WFEnt(X_f; G_j) \tag{4.10}$$

. The fuzzy entropy $FEnt(G_j)$ and the weighted fuzzy entropy $WFEnt(X_f; G_j)$ are determined as:

$$FEnt(G_j) = \sum_{m=1}^{M} -\frac{|G_{j,C_m}|}{|G_j|} log_2(\frac{|G_{j,C_m}|}{|G_j|}) \tag{4.11}$$

$$WFEnt(G_j) = \sum_{m=1}^{M} \frac{|G_j|}{|G|} FEnt(G_j) \quad (4.12)$$

respectively, where G_{j,C_m} is the set of examples in G_j with class label equal to C_m.

In case of categorical variables, we split the parent node into a number of child nodes equal to the number of possible values for the variable. Cardinality is computed as follows:

$$|G_j| = \sum_{i=1}^{N_l} TN(1, \mu_G(\mathbf{x}_i)) \quad (4.13)$$

where G_j corresponds to the subset of the jth categorical value. Note that a variable can be considered only once in the same path from the root to the leaf.

Algorithm 2 details the pseudo code of the multi-way splitting to generate child nodes for a given parent node PN.

Algorithm 2 Pseudo code of multi-way splitting applied to a given parent node.

Require: Let PN be the parent node on which performing the multi-way splitting, and G be the subset of examples which belong to PN

```
 1: procedure MULTISPLITTINGNODE(in: PN, G)
 2:     for each attribute X_f in X do
 3:         if X_f is continuous then
 4:             compute FGain by using Eq. 4.10 for cardinality
 5:         else
 6:             compute FGain by using Eq. 4.13 for cardinality
 7:         end if
 8:     end for
 9:     f̂ ← get attribute with the highest value of FGain
10:     children ← create an empty list
11:     for each A_{f̂,j} defined in f̂ do
12:         G_j ← get points from G belonging only to A_{f̂,j}
13:         child ← create node by using A_{f̂,j} and G_j
14:         children ← insert child
15:     end for
16:     return children
17: end procedure
```

Unlike FMDT, FBDT performs binary splitting at each node. As shown in Figure 4.4, the algorithm generates exactly 2 child nodes. To calculate the split with the maximum $FGain$, we exploit all possible candidates, by grouping together adjacent fuzzy sets into two disjoint groups Z_1 and Z_2. The two subsets of examples, G_1 and G_2, contain the points which belong to the support of the fuzzy sets contained in Z_1 and Z_2, respectively. A fuzzy partition with T_f fuzzy sets generates $T_f - 1$ candidates. Starting with $Z_1 = \{A_{f,1}\}$ and $Z_2 = \{A_{f,2}, ..., A_{f,T_f}\}$, we compute the fuzzy information gain by applying Eq. 4.11 and Eq. 4.12, where $j = \{1, 2\}$, and G_1 and G_2 contain only the examples which

belong to the support of fuzzy sets in Z_1 and Z_2, respectively. Iteratively, the algorithm investigates all candidates by moving the first fuzzy set in Z_2 to Z_1 and computing the corresponding $FGain$, until $Z_2 = \{A_{f,T_f}\}$. The pair (Z_1,Z_2), which obtains the highest $FGain$, is used for creating the two child nodes. The two nodes contain, respectively, the examples that belong to the support of the fuzzy sets in Z_1 and Z_2.

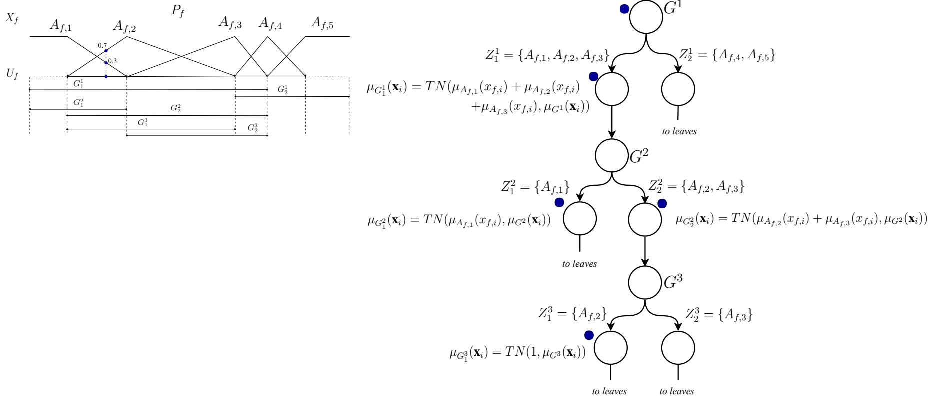


Figure 4.4: An example of binary split performed by FBDT on a continuous attribute partitioned by five triangular fuzzy sets.

In case of categorical variables, FBDT still performs binary splits. However, since a categorical attribute with L values generates $2^{L-1} - 1$ candidates, the computational cost can become very prohibitive for a large number of values. In case of binary classification, we can reduce the number of candidates to $L - 1$ by sorting the categorical values according to the probability of membership to the positive class. As proved in [24] and [159], this approach gives the optimal split in terms of entropy. In case of multiclass classification, we adopt the heuristic method proposed in [118] to approximate the best split: the number of candidates is reduced to $L - 1$ by sorting the categorical values according to their impurity.

In FBDT, both categorical and continuous attributes can be considered in several fuzzy decision nodes in the same path from the root to a leaf. In each node, we apply the same binary splitting approach described above but restricted only to the categorical values or fuzzy sets considered in the node. Figure 4.4 shows the splitting approach performed by FBDT, considering the same fuzzy partition used for FMDT. Let us suppose that, at the root, the attribute X_f is selected. Further, let us assume that the two child nodes of the root node contain instances belonging to the supports of $Z_1^1 = \{A_{f,1}, A_{f,2}, A_{f,3}\}$ and $Z_2^1 = \{A_{f,4}, A_{f,5}\}$, respectively. If X_f is selected again in the path starting from Z_1 , then the two child nodes are created by considering only the three fuzzy sets in Z_1^1 and the instances contained in $[a_{f,1}, c_{f,3}]$, where $a_{f,1}$ and $c_{f,3}$ are the lower and upper bounds of the supports of $A_{f,1}$ and $A_{f,3}$, respectively. If the

highest fuzzy information gain is obtained by splitting the three fuzzy sets into $\{A_{f,1}\}$ and $\{A_{f,2}, A_{f,3}\}$, then the two child nodes contain the instances belonging to the intervals $[a_{f,1}, c_{f,1}]$ and $[a_{f,2}, c_{f,3}]$, respectively.

Due to the use of the T-norm, and in particular of the product employed in our experiments, the binary splitting approach tends to penalize the cardinality of continuous attributes that are repeatedly selected along a same path. To limit this effect, we use a strategy that keeps track of the fuzzy sets, which have been activated by an instance in the path from the root to the leaves: we consider the membership value to a fuzzy set only the first time the fuzzy set is met. The subsequent times the membership value is set to 1 in the computation of the T-norm. For example, let us suppose that an instance $\mathbf{x}_i$ belongs to fuzzy sets $A_{f,1}$ and $A_{f,2}$ with membership values 0.3 and 0.7, respectively, as shown in Figure 4.4 (see blue circle). When splitting G^2, the instance contributes to the cardinality computation of G_1^2 and G_2^2 with $\mu_{G_1^2}(\mathbf{x}_i) = TN(\mu_{A_{f,1}}(\mathbf{x}_i), \mu_{G^2}(\mathbf{x}_i))$ and $\mu_{G_2^2}(\mathbf{x}_i) = TN((\mu_{A_{f,2}}(\mathbf{x}_i)), \mu_{G^2}(\mathbf{x}_i))$, respectively. When splitting G^3, the membership degree $\mu_{A_{f,2}}(\mathbf{x}_i)$ of the instance $\mathbf{x}_i$ to $A_{f,2}$ is considered equal to 1 and the instance contributes to the cardinality computation of the subset G_1^3 with $\mu_{G_1^3}(\mathbf{x}_i) = TN(1, \mu_{G^3}(\mathbf{x}_i))$. On the other hand, the actual fuzzy membership value $\mu_{A_{f,2}}(\mathbf{x}_i)$ of instance $\mathbf{x}_i$ to $A_{f,2}$ has been already considered in the computation of $\mu_{G^3}(\mathbf{x}_i)$. In general, cardinality of G_z with $z = 1, 2$ is computed as:

$$|G_z| = \sum_{i=1}^{N_z} \mu_{G_z}(\mathbf{x}_i) \tag{4.14}$$

where N_z is the number of instances in G_z and $\mu_{G_z}(\mathbf{x}_i)$ is the membership degree of $\mathbf{x}_i$ to set G_z calculated as described above. For categorical attributes, the cardinality can still be computed with Eq. 4.14, where $\forall \mathbf{x}_i \in TR$, $\mu_{G_z}(\mathbf{x}_i) = TN(1, \mu_G(\mathbf{x}_i))$.

Algorithm 3 details the pseudo code of the binary splitting approach for generating two child nodes from a given parent node PN.

Unlike crisp decision trees, for both FMDT and FBDT, we label each leaf node LN with all the classes that have at least one example in the leaf node. Each class C_m has an associated weight w_m^{LN} proportional to the fuzzy cardinality of training instances of that m^{th} class in the node. More formally, w_m^{LN} is computed as:

$$w_m^{LN} = \frac{|G_{C_m}|}{|G|} \tag{4.15}$$

where G_{C_m} is the set of instances in G with class label equal to G_m.

Both FMDT and FBDT adopt the weighed vote for deciding the class to be output for the unlabeled instance. For each class, the vote is computed as sum of the activation degrees determined by any leaf node of the tree for that class, where the activation degree is calculated by Eq. 4.2 . In case of FBDT, the fuzzy cardinality used in the computation of the matching degree is determined by considering the membership value to a specific fuzzy set only one time, also if the fuzzy set is met more times in the path from the root to the leaf, as explained above. Each activated leaf produces a list of class association de-

Algorithm 3 Pseudo code of binary-splitting approach given a parent node.

Require: Let PN be a parent node on which performing the binary splitting, and let G be the subset of instances which belong to PN

1: **procedure** BINARYSPLITTINGNODE(**in**: PN, S)
2:     **for** each attribute X_f in **X do**
3:         **for** each *candidate binary split* in X_f **do**
4:             compute $FGain$ by using the cardinality expressed by Eq. 4.14
5:         **end for**
6:     **end for**
7:     $(Z_1, Z_2) \leftarrow$ get the split with the highest value of $FGain$
8:     $children \leftarrow$ create empty list
9:
10:     /*** *Create left child node* ***/
11:     $G_1 \leftarrow$ get points from G belonging to Z_1
12:     $child_1 \leftarrow$ create node by using Z_1 and G_1
13:     /*** *Create right child node* ***/
14:     $G_2 \leftarrow$ get points from G belonging to Z_2
15:     $child_2 \leftarrow$ create node by using Z_2 and G_2
16:
17:     $children \leftarrow$ insert $child_1$ and $child_2$
18:     **return** $children$
19: **end procedure**

grees, which are summed up to compute the strength of vote for that class, as described in Eq. 4.2. The unlabeled pattern $\widehat{x}$ is associated with the class with the highest strength of vote.

4.2.3 The Distributed Approach

In Section 4.1 we have pointed out that the current implementations of FDTs are not suitable for managing big data. In this section we describe our distributed approach by describing in detail the distributed implementation of the two main steps of the overall algorithm, namely *Fuzzy Partitioning* and *FDT Learning*. We highlight that our approach is based on the Map-Reduce paradigm and can be easily deployed on several cloud-computing environments such as Hadoop, Flink and Spark.

Let V be the number of chunks used for splitting the training set and Q the number of CUs available in the cluster. Each chunk fed only one Map task, while one CU can process several tasks, both Map and Reduce. Obviously, only Q tasks can be executed in parallel.

The distributed implementation of the fuzzy partitioning approach described in Section 4.2.1 is similar to the one we have proposed in Section 3.5.1. In particular, the approach described in Section 4.2.1 is not suitable for dealing with a huge amount of data because both the sorting of the values and the computation of fuzzy information gain for each possible candidate fuzzy partition are computationally expensive in case of datasets

with millions or even billion of instances. To overcome this drawback, we adopt an approximation of FPFE by limiting the number of possible candidate partitions to be analyzed. In particular, for each single chunk of the training set, independently of the others, we apply the sorting of the values and split the domain of the continuous attributes into a fixed number of equi-frequency bins. Then, we aggregate the lists of the bin boundaries generated for each chunk and, for each pair of consecutive bin boundaries, we generate a new bin and compute the distribution of the classes among the instances belonging to the bin. Finally, we generate candidate fuzzy partitions for each bin boundary and exploit the class distribution in each bin for computing the fuzzy information gain at each iteration of the algorithm. Obviously, the lower the number of bins used for splitting the domain of the attribute is, the coarser the approximation in determining the fuzzy partition is.

Figure 4.5 shows the overall Fuzzy Partitioning process. In the first Map-Reduce step, the algorithm determines the bin boundaries. In the map phase, each CU loads a chunk of the training set and, for each attribute, sort the values and computes the bin boundaries by splitting the data according to a fixed number of equi-frequency bins. Then, in the reduce phase, for each attribute, all the bin boundaries are grouped together and sorted. Thus, the output of the first Map-Reduce step is a sorted list of bin boundaries for each attribute. The second Map-Reduce step defines a Ruspini fuzzy partition for each attribute. In the map phase, each Map task computes, for each bin determined by consecutive bin boundaries, the percentage of instances belonging to the different classes. In the reduce phase, each reduce task generates a fuzzy partition for a specific attribute, as described in Section 4.2.1, using the bin boundaries for defining *candidate fuzzy partitions* and the distribution of the classes in the bin for computing the fuzzy information gain. The proposed distributed approach can manage a large number of instances: the bin boundaries allow reducing the number of candidate fuzzy partitions to be explored. Obviously, the number of equi-frequency bins is a parameter of the approach, which affects both the fuzzy partitioning of the continuous attributes and the results of the FDT. However, this parameter is not particularly critical. Indeed, we have to consider that we are managing millions of data. Thus, a difference of a few instances in determining the best fuzzy partition is generally negligible in terms of the accuracy achieved by the FDTs.

The first Map-Reduce phase scans the training set to compute at most $\Omega = V \cdot \Gamma$ bin boundaries, where $\Gamma = 100/\gamma + 1$ is the number of bin boundaries per chunk. This value depends on the percentage γ of the vth chunk size. In our experiments, we set $\gamma = 0.1\%$. Algorithm 4 details the pseudo code of the first Map-Reduce phase.

Each Map-Task, first, loads the vth chunk of the training set, and then for each continuous attribute X_f, computes and outputs the bin boundaries of equi-frequency bins, where each bin contains a number of instances equal to the percentage γ of the data chunk. Let $BB_{v,f} = \{b_{v,f}^{(1)}, ..., b_{v,f}^{(\Gamma)}\}$ be the sorted list of bin boundaries for the fth attribute extracted from the vth chunk. The Map-Task outputs a key-value pair $\langle key = f, value = BB_{v,f} \rangle$, where f is the index of the fth attribute. Each Reduce-Task is fed by V lists $List(BB_{v,f})$ and, for the fth attribute, outputs $\langle key = f, value = BB_f \rangle$, where $BB_f = \{b_f^{(1)}, ..., b_f^{(\Omega)}\}$ with, $\forall w \in [1, ..., \Omega - 1]$, $b_f^{(w)} < b_f^{(w+1)}$ is the sorted list of

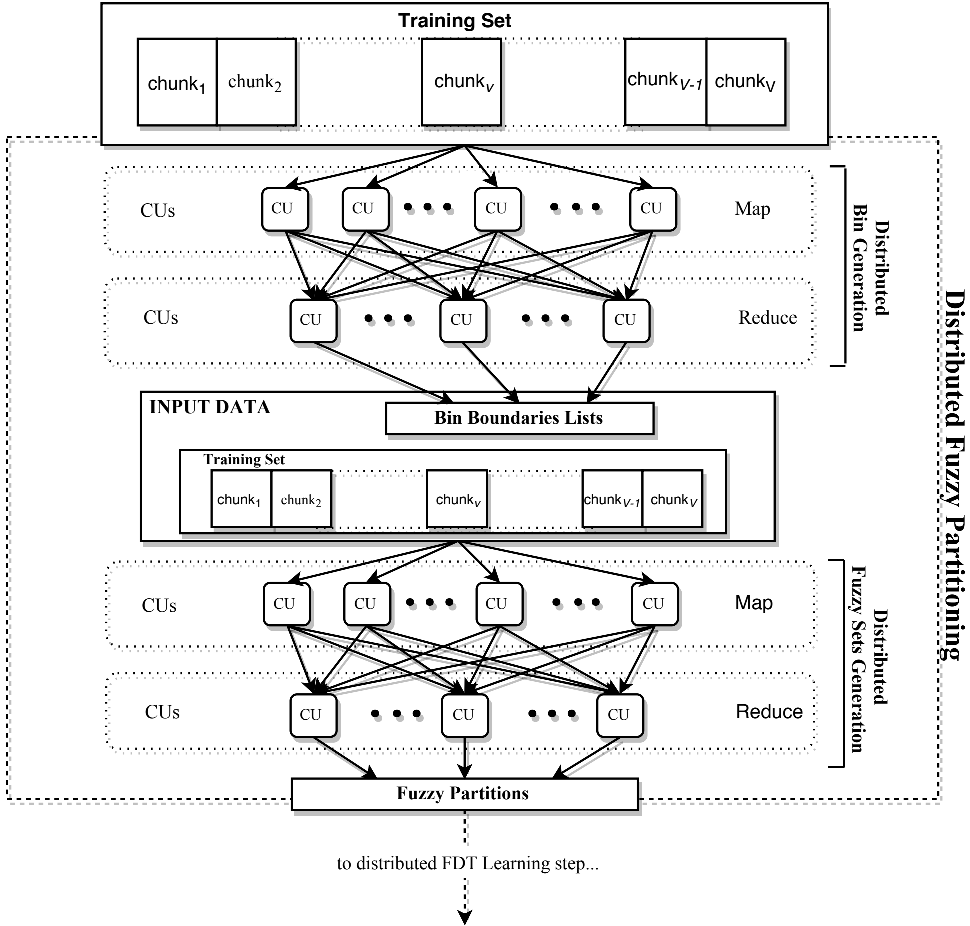


Figure 4.5: The overall distributed Fuzzy Partitioning of the FDT.

Algorithm 4 Pseudo code of the first Map-Reduce phase of distributed Fuzzy Partitioning.

Require: TR split into V chunks. In the following, each chunk is denoted as $chunk_v$.

1: **procedure** MAP-TASK(**in**: $chunk_v$, γ)
2: **for** each continuous attribute X_f in X **do**
3: sort values of X_f
4: $BB_{v,f} \leftarrow$ compute boundaries of equi-frequency bins according to γ
5: output $\langle key = f, value = BB_{v,f} \rangle$
6: **end for**
7: **end procedure**
8: **procedure** REDUCE-TASK(**in**: f, $List(BB_{v,f})$)
9: $BB_f \leftarrow$ group and sort elements of $List(BB_{v,f})$
10: output $\langle key = f, value = BB_f \rangle$
11: **end procedure**

the bin boundaries for attribute X_f. Space and time complexities, for the Map phase, are $O(\lceil \frac{V}{Q} \rceil \cdot N/V)$ and $O(\lceil \frac{V}{Q} \rceil \cdot (F \cdot N \cdot (log(N/V))/V))$, respectively. For the Reduce phase, space and time complexities are $O(F \cdot \Omega/Q)$ and $O(F \cdot (\Omega \cdot log(\Omega))/Q)$, respectively.

The second Map-Reduce step scans the training set again: in the map phase, for each chunk of the training set, it computes the percentage of instances belonging to each class for each bin, and in the reduce phase it generates the fuzzy partition. Algorithm 5 details the pseudo code of the second Map-Reduce phase.

Algorithm 5 Pseudo code of the second Map-Reduce step of the distributed Fuzzy Partitioning.

Require: TR split into V chunks (in the following, each chunk is denoted as $chunk_v$). Matrix BB that contains for each row a list of sorted bin boundaries BB_f.
1: **procedure** MAP-TASK(**in**: $chunk_v$, BB, M)
2: $W_{v,f} \leftarrow$ create F vectors according to each BB_f and M
3: **for** each instance $\mathbf{x}_n$ in $chunk_v$ **do**
4: **for** each continuous attribute X_f in **X do**
5: $W_{v,f} \leftarrow$ update number of instances belonging to each class
6: **end for**
7: **end for**
8: **for** each continuous attribute X_f in **X do**
9: output $\langle key = f, value = W_{v,f} \rangle$
10: **end for**
11: **end procedure**
12: **procedure** REDUCE-TASK(**in**: f, $List(W_{v,f})$, BB_f)
13: $W_f \leftarrow$ element-wise addition of $List(W_{v,f})$
14: $P_f \leftarrow$ compute Fuzzy Partitioning with W_f and B_f
15: output $\langle key = f, value = P_f \rangle$
16: **end procedure**

Each Map-Task, first, loads the vth chunk of the training set and, for each attribute X_f, initializes a vector $W_{v,f}$ of $\Omega - 1$ elements. Each element $W_{v,f}^{(r)}$ corresponds to the bin $(b_f^r, b_f^{(r+1)}]$ and contains a vector of M elements, which stores, for each of the M classes, the number of instances of the class belonging to the rth bin in the vth chunk. Then, for each instance of the chunk, the Map-Task updates $W_{v,f}$ and finally outputs a key-value pair $\langle key = f, value = W_{v,f} \rangle$. Each Reduce-Task is fed by a list $List(W_{v,f})$ of V vectors. For each attribute X_f, it first creates a vector W_f of $\Omega - 1$ elements by performing an element-wise addition of all V vectors $W_{v,f}$. Thus, W_f stores the number of instances for each class in each bin along the overall training set. Then, the Reduce-Task applies the Fuzzy Partitioning as described in Section 4.2.1, where *candidate fuzzy partitions* are defined upon bin boundaries and the fuzzy mutual information is computed according to W_f. Finally, it outputs the key-pair $\langle key = f, value = P_f \rangle$, where P_f is the Ruspini fuzzy partition defined on the fth attribute. Space and time complexities of the Map phase are $O(\lceil \frac{V}{Q} \rceil \cdot N/V)$ and $O(\lceil \frac{V}{Q} \rceil \cdot (N \cdot log(\Omega)/V))$, respectively. For the Reduce

phase, space and time complexities are $O(F \cdot (\Omega - 1)/Q)$ and $O(F \cdot (max(|TR_f|) \cdot (\Omega - 1)^2)/Q)$, respectively, where $|TR_f|$ is the number of intervals generated recursively by the Fuzzy Partitioning for retrieving the best core of triangular fuzzy sets in each interval $TR_{f,p}$ for the fth attribute. Since for each attribute, fuzzy partitioning generates a different number of intervals (in case no fuzzy set is defined on attribute X_f, $|TR_f| = 1$), the time complexity is upper bounded by the maximum value $max(|TR_f|)$.

As regards the DFDT learning, in order to manage a large amount of data, we distribute the computation of the best split for each node across the CUs. Figure 4.6 illustrates the overall DFDT learning algorithm. The algorithm executes iteratively a Map-Reduce step which operates on the list of nodes to be split. At each iteration, a group of nodes is retrieved from the list and is processed as follows: first, in the map phase, a scan over the training set is performed for collecting the necessary statistics for each node of the set; then, in the reduce phase, the statistics of each node are aggregated together and the best split for a given node is computed by each CU; finally the FDT and the list of nodes are updated for the next iteration. Note that, at each iteration, the considered nodes are the deepest ones in the branches of the tree.

The proposed distributed approach allows managing a large amount of data: performing the splitting on a group of nodes significantly reduces the number of scans over the training set, but also requires a larger quantity of memory and a longer computation time for each iteration (the computational cost is limited by collecting and aggregating the necessary statistics). Thus, the maximum number of nodes, which can be processed in parallel at each iteration, depends on the memory availability on the cluster. Obviously, the higher the number of categorical values and fuzzy sets defined by the fuzzy partitioning, the higher the memory used for collecting the statistics for each attribute and the lower the number of nodes that can be processed in parallel at each iteration.

Algorithm 6 details the pseudo code of the DFDT learning.

More formally, let H be the number of iterations performed by the algorithm and h be the index of the hth iteration. The proposed approach first initializes a list of nodes R with only one element consisting of the root of the tree and then iteratively retrieves a group R_h of Y nodes from R, where $Y = min(size(R), maxY)$ is computed according to the number of nodes in R and a fixed threshold $maxY$, which defines the number of nodes processed at most at each iteration. Finally, it performs a Map-Reduce step for distributing the growing process of the tree. The vth Map-Task, first, loads the vth chunk of the training set and then, for each node NT_y in R_h, initializes a vector $D_{v,y}$ of $|D| = \sum_{\forall f \in F} T_f$ instances. We recall that if X_f is continuous, then T_f is the number of fuzzy sets defined by the Fuzzy Partitioning process; otherwise, if X_f is categorical, then T_f is number of categorical values. For each attribute of each instance of the chunk, the Map-Task updates all $D_{v,y}$ vectors by exploiting Eq. 4.9 or Eq. 4.13 in case the attribute is continuous or categorical, respectively, and then, for each node, outputs the key-value pair $\langle key = y, value = D_{v,y} \rangle$, where y is the index of the yth node in R_h. At the end of the map phase, each element of $D_{v,y}$ stores the cardinality of each attribute value from the root to NT_y only for the instances in the vth chunk. Each Reduce-Task is fed by

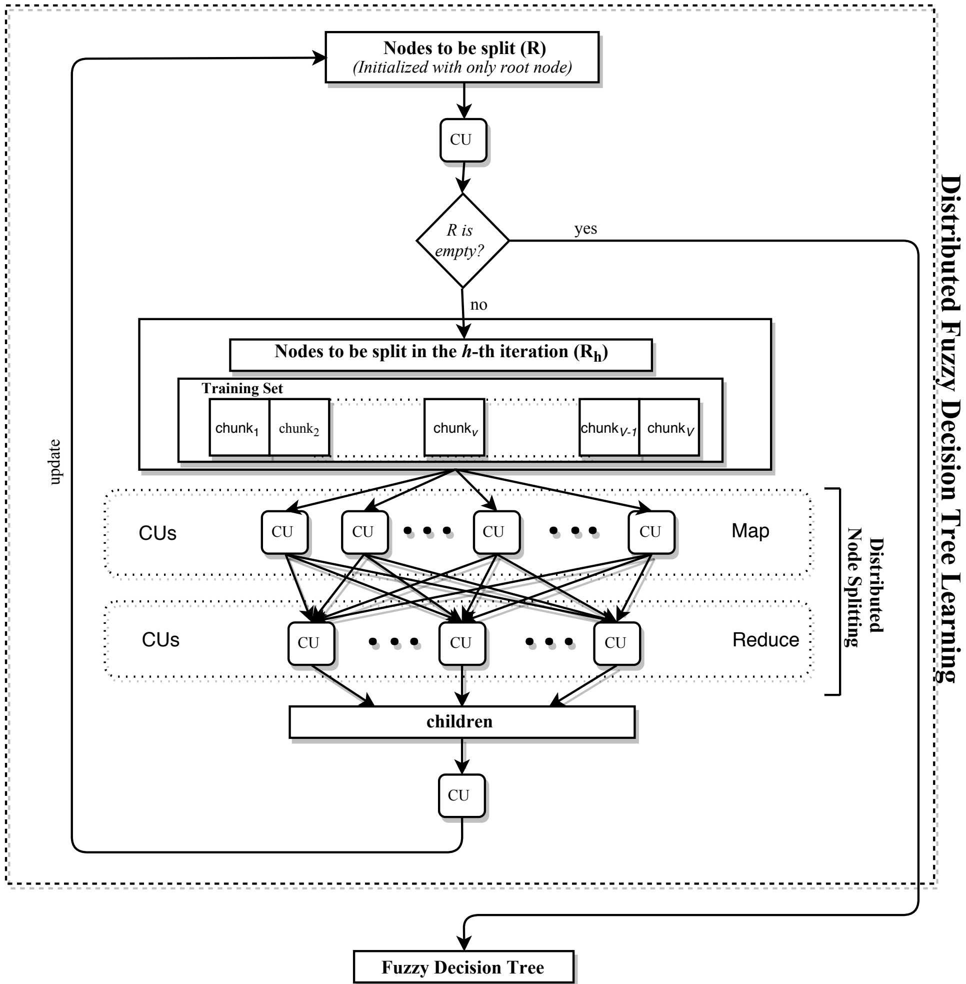


Figure 4.6: The overall DFDT Learning approach.

a list, say $List(D_{v,y})$, of vectors $D_{v,y}$ and first it creates a vector D_y by performing an element-wise addition of all V vectors in $List(D_{v,y})$. Thus, D_y stores the cardinality of each attribute value from the root to NT_y along the overall training set. Then, the Reduce-Task generates and outputs the child nodes by employing Algorithm 2 or Algorithm 3 in case of multi-way or binary splitting methods, respectively. The children generated from each NT_y are finally used to update the tree and R: if a child node is not labeled as leaf, then it is inserted into the list and employed at the next iterations. The algorithm repeats all the steps until R is empty. Space and time complexities of the Map phase are $O(\lceil \frac{V}{Q} \rceil \cdot N/V)$ and $O(\lceil \frac{V}{Q} \rceil \cdot (N \cdot Y \cdot log(|D|)/V))$, respectively. For the Reduce phase, space and time complexities are $O(Y/Q)$ and $O(Y \cdot |allSplits|/Q)$, respectively, where $|allSplits|$ is the number of splits that have to be investigated for computing the best split

Algorithm 6 Pseudo code of the DFDT Learning.

Require: TR split into V chunks, splitting method $splitMet$, stopping method $stopMet$. In the following, each chunk is denoted as $chunk_v$.

1: **procedure** FDTLEARNING(**in**: TR, $splitMet$, $stopMet$, $maxY$)
2: $tree \leftarrow$ create $root$
3: $R \leftarrow$ create list and insert $root$
4: **repeat**
5: $R_h \leftarrow$ get $min(size(R), maxY)$ nodes from R
6: $children \leftarrow$ call Map-Reduce Tasks
7: **for** each $child$ in $children$ **do**
8: $tree \leftarrow$ update model with $child$
9: **if** ISNOTLEAF($child$, $stopMet$) **then**
10: $R \leftarrow$ insert $child$
11: **end if**
12: **end for**
13: **until** R is not empty
14: **return** $tree$
15: **end procedure**
16: **procedure** MAP-TASK(**in**: $chunk_v$, R_h)
17: **for** each node NT_y in R_h **do**
18: $D_{v,y} \leftarrow$ create a vector of $|D|$ elements
19: **for** each instance $\mathbf{x}_n$ in $chunk_v$ **do**
20: $D_{v,y} \leftarrow$ update statistics with $x_{f,n}$ according to Eq. 4.9 or Eq. 4.13
21: **end for**
22: output $\langle key = y, value = D_{v,y} \rangle$
23: **end for**
24: **end procedure**
25: **procedure** REDUCE-TASK(**in**: y, $List(D_{v,y})$)
26: $D_y \leftarrow$ element-wise addition of $List(D_{v,y})$
27: **if** $splitMet$ is multiple splitting **then**
28: $children \leftarrow$ MULTISPITTINGNODE(NT_y, D_y) (See Algorithm 2)
29: **else**
30: $children \leftarrow$ BINARYSPITTINGNODE(NT_y, D_y) (See Algorithm 3)
31: **end if**
32: output $\langle key = y, value = children \rangle$
33: **end procedure**

among all attribute for the node. Note that $|allSplits| = F$ and $|allSplits| = |D|$ for multi-way and binary splitting approaches, respectively. Since time complexity of Map phase represents the heaviest part of computational cost, the time complexity of Algorithm 6 is $O(H \cdot (\lceil \frac{V}{Q} \rceil \cdot (N \cdot log(|D|)/V)))$.

4.3 Experimental Study

We performed several experiments for investigating the behavior of the proposed approach, focusing on the performance in terms of classification accuracy, model complexity and execution time.

As shown in Table 4.1, we employed 8 well-known big datasets freely available from UCI[2] and LIBSVM[3] repositories. The datasets are characterized by different numbers of input/output instances (from 1 million to 11 millions), classes (from 2 to 23), and attributes (from 10 to 41). For each dataset, we also report the number of numeric (*num*) and categorical (*cat*) attributes.

Table 4.1: Big datasets used in the experiments.

Dataset	# Instances	# Attributes	# Classes
ECO (ECO)	4,178,504	16 (num:16)	10
EME (EME)	4,178,504	16 (num:16)	10
Higgs (HIG)	11,000,000	28 (num:28)	2
KDDCup 1999 2 Classes (KDD99_2)	4,856,151	41 (num:26, cat:15)	2
KDDCup 1999 5 Classes (KDD99_5)	4,898,431	41 (num:26, cat:15)	5
KDDCup 1999 (KDD99)	4,898,431	41 (num:26, cat:15)	23
Poker-Hand (POK)	1,025,010	10 (cat:10)	10
Susy (SUS)	5,000,000	18 (num: 18)	2

All the experiments have been executed on a cluster consisting of one master equipped with a 4-core CPU (Intel Core i5 CPU 750 x 2.67 GHz), 8 GB of RAM and a 500GB Hard Drive, and three slave nodes equipped with a 4-core CPU with Hyperthreading (Intel Core i7-2600K CPU x 3.40 GHz, 8 threads), 16GB of RAM and a 1 TB Hard Drive. All nodes are connected by a Gigabit Ethernet (1 Gbps) and run Ubuntu 12.04. The algorithm has been deployed upon Apache Spark 1.5.2 as data-processing framework: the master hosts the *driver program*, while each slave runs an *executor*. The training sets are stored in the HDFS.

4.3.1 Performance analysis

In this section, we analyze the performance of both FMDT and FBDT in terms of accuracy, model complexity, and execution time and compare both of them with the distributed implementation of a Distributed Decision Tree (DDT) available in MLlib [140]. DDT performs a recursive binary partitioning of the attribute space. The partitions of the continuous attributes are generated by dividing each attribute into equi-frequency bins (at most $maxBins$) over a sampled fraction of the data. Then, at each decision node, the best split is chosen by selecting the one that maximizes the information gain. Entropy or Gini index can be used for computing impurity of the node. Further, a maximum depth $maxDepth$ of the tree can be fixed by the user.

[2] Available at `https://archive.ics.uci.edu/ml/datasets.html`
[3] Available at `https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/`

Table 4.2 summarizes, for each algorithm, the parameters used in the experiments. For FMDT, we limit the number of fuzzy sets defined for each attribute during the fuzzy partitioning process and the number of instances belonging to each node. by setting $\phi = 2\%$ and $\lambda = 0.01\%$, respectively. In particular, we force that the support of each fuzzy set contains at least $\phi = 0.02 \cdot N$ instances and the number of instances for each node is $\lambda = 0.00001 \cdot N$. We have performed different experiments varying ϕ from $0.01 \cdot N$ to $0.1 \cdot N$ with step $0.01 \cdot N$, and λ from $0.00001 \cdot N$ to $0.01 \cdot N$, with step $0.00001 \cdot N$. We have observed that the best accuracy is achieved with $\phi = 0.02 \cdot N$ and $\lambda = 0.00001 \cdot N$. In practice, we have verified that smaller supports tend to fragment the data too quickly, leaving insufficient instances at the deepest nodes of the tree. After a limited number of levels, it is unlikely to perform further splits. On the other hand, wider supports do not allow obtaining satisfactory fuzzy partitions. Further, higher and lower values of λ lead to a classifier, respectively, excessively general and specialized on the training set, penalizing the performance on the test set. Also, lower values for ϕ and λ increase the overall runtime with no real advantage.

Binary splitting overcomes the previous discussed drawbacks. Thus, for FBDT no specific limitation is imposed and we set $\phi = \lambda = 1$ instance. For both FMDT and FBDT, we set $\gamma = 0.1\%$ as suggested by authors in [17] and $TNorm = product$. As regard DDT, we adopted the values suggested in the guidelines provided with the library.

Table 4.2: Values of the parameters for each algorithm used in the experiments.

Method	Parameters
FMDT	$\gamma = 0.1\%, \phi = 0.02 \cdot N, \lambda = 0.00001 \cdot N, TNorm = product$
FBDT	$\gamma = 0.1\%, \phi = 1, \lambda = 1, TNorm = product$
DDT	$maxBins = 32, Impurity = Entropy$

For each dataset and for each algorithm, we performed a five-fold cross-validation by using the same folds for all the datasets and varying the maximum depth β of the tree. Table 4.3 shows, for each dataset and for each algorithm, the average values $\pm$ standard deviation of the accuracy, both on the training ($AccTr$) and test sets ($AccTs$) obtained by the algorithms. The highest accuracy values for each dataset are shown in bold. Table 4.4 shows the complexity of each algorithm. For each experiment, we report the number of nodes ($\#Nodes$), the number of leaves ($\#Leaves$) and the minimum ($minDpt$), the maximum ($maxDpt$) and average ($avgDpt$) depths of the trees.

The analysis of the three tables highlights that, on average, both FMDT and FBDT outperform DDT in all datasets. As regards FDTs, we can observe that, when comparing trees with the same depth, the multi-way splitting tends to achieve higher accuracy because it is able to investigate a higher number of correlations between attributes by generating a higher number of nodes at each level. On the other hand, as shown in Table 4.4, the trees are characterized by a significantly higher number of nodes, increasing the complexity. For instance, for ECO, EME, HIG and SUS, FMDT employs more than

Table 4.3: Average accuracy $\pm$ standard deviation achieved by FMDT, FBDT and DDT.

		FMDT		FBDT		DDT	
Dataset	β	Acc_{Tr}	Acc_{Ts}	Acc_{Tr}	Acc_{Ts}	Acc_{Tr}	Acc_{Ts}
ECO	5	$\mathbf{97.641} \pm 0.019$	$\mathbf{97.585} \pm 0.041$	78.244 ± 0.015	78.242 ± 0.037	77.718 ± 0.765	77.721 ± 0.729
	10	–	–	89.347 ± 0.105	89.335 ± 0.142	88.099 ± 0.164	88.082 ± 0.179
	15	–	–	97.315 ± 0.025	97.262 ± 0.045	95.874 ± 0.269	95.756 ± 0.286
EME	5	96.962 ± 0.008	96.913 ± 0.018	77.381 ± 0.245	77.354 ± 0.303	77.270 ± 1.569	77.254 ± 1.592
	10	–	–	90.751 ± 0.051	90.705 ± 0.051	89.756 ± 0.171	89.729 ± 0.186
	15	–	–	$\mathbf{96.991} \pm 0.021$	$\mathbf{96.928} \pm 0.032$	95.856 ± 0.203	95.726 ± 0.194
HIG	5	72.638 ± 0.018	71.253 ± 0.029	66.451 ± 0.013	66.441 ± 0.025	66.344 ± 0.080	66.335 ± 0.106
	10	–	–	70.723 ± 0.013	70.697 ± 0.022	70.481 ± 0.040	70.403 ± 0.063
	15	–	–	72.631 ± 0.019	$\mathbf{72.266} \pm 0.008$	$\mathbf{73.073} \pm 0.031$	71.871 ± 0.013
KDD99_2	5	99.986 ± 0.006	99.986 ± 0.005	99.989 ± 0.000	99.987 ± 0.000	99.980 ± 0.008	99.979 ± 0.008
	10	–	–	99.999 ± 0.000	$\mathbf{99.999} \pm 0.000$	99.999 ± 0.001	$\mathbf{99.999} \pm 0.001$
	15	–	–	99.999 ± 0.000	$\mathbf{99.999} \pm 0.000$	$\mathbf{100.000} \pm 0.000$	$\mathbf{99.999} \pm 0.000$
KDD99_5	5	99.976 ± 0.002	99.973 ± 0.003	99.893 ± 0.000	99.894 ± 0.002	99.669 ± 0.010	99.882 ± 0.010
	10	–	–	99.995 ± 0.000	99.992 ± 0.001	99.991 ± 0.001	99.989 ± 0.001
	15	–	–	$\mathbf{99.999} \pm 0.000$	$\mathbf{99.995} \pm 0.000$	$\mathbf{99.999} \pm 0.001$	99.994 ± 0.001
KDD99	5	99.950 ± 0.001	99.948 ± 0.002	99.597 ± 0.008	99.598 ± 0.009	99.669 ± 0.104	99.669 ± 0.103
	10	–	–	99.990 ± 0.000	99.971 ± 0.001	99.991 ± 0.001	99.989 ± 0.001
	15	–	–	99.997 ± 0.000	$\mathbf{99.994} \pm 0.001$	$\mathbf{99.999} \pm 0.000$	99.993 ± 0.001
POK	5	$\mathbf{78.479} \pm 0.031$	$\mathbf{77.176} \pm 0.068$	54.708 ± 0.405	54.696 ± 0.432	54.708 ± 0.405	54.696 ± 0.432
	10	–	–	58.806 ± 0.508	58.490 ± 0.599	58.806 ± 0.508	58.490 ± 0.599
	15	–	–	67.553 ± 0.422	62.479 ± 0.504	67.553 ± 0.422	62.479 ± 0.504
SUS	5	$\mathbf{80.962} \pm 0.007$	79.639 ± 0.016	77.312 ± 0.060	77.230 ± 0.057	77.023 ± 0.025	77.018 ± 0.038
	10	–	–	79.118 ± 0.016	79.091 ± 0.024	79.022 ± 0.043	78.940 ± 0.052
	15	–	–	79.969 ± 0.030	$\mathbf{79.722} \pm 0.043$	80.393 ± 0.026	79.304 ± 0.032

Table 4.4: Complexities of FMDT, FBDT and DDT.

		FMDT			FBDT			DDT		
Dataset	β	$\#Nodes$	$\#Leaves$	$maxDpt$	$\#Nodes$	$\#Leaves$	$maxDpt$	$\#Nodes$	$\#Leaves$	$maxDpt$
ECO	5	222,694	200,048	5	63	32	5	63	32	5
	10	-	-	-	1,695	849	10	1,530	765	10
	15	-	-	-	17,532	8,741	15	12,323	6,162	15
EME	5	240,406	218,557	5	63	32	5	63	32	5
	10	-	-	-	1,694	847	10	1,521	761	10
	15	-	-	-	20,996	10,477	15	14,515	7,258	15
HIG	5	972.779	920,942	5	63	32	5	63	32	5
	10	-	-	-	1,686	844	10	2,045	1,023	10
	15	-	-	-	34,444	17,209	15	49,822	24,911	15
KDD99_2	5	703	630	5	41	21	5	37	19	5
	10	-	-	-	131	66	10	95	48	10
	15	-	-	-	222	112	15	121	61	15
KDD99_5	5	2,716	2,351	5	46	24	5	49	25	5
	10	-	-	-	335	168	10	356	179	10
	15	-	-	-	779	389	15	544	272	15
KDD99	5	2,164	1,875	5	37	19	5	40	20	5
	10	-	-	-	369	185	10	303	152	10
	15	-	-	-	972	485	15	581	291	15
POK	5	30,940	28,561	4	63	32	5	63	32	5
	10	-	-	-	2,024	1,012	10	2,024	1,012	10
	15	-	-	-	44,297	22.149	15	44,297	22,149	15
SUS	5	805,076	758,064	5	63	32	5	63	32	5
	10	-	-	-	1,360	681	10	1,984	993	10
	15	-	-	-	21,452	10,723	15	35,133	17,567	15

200,000 leaves with only five levels of depth. For higher values of β the algorithm generates too many nodes and the overall process takes an unreasonable amount of time. For this reason, no result for higher values of β has been reported in Table 4.3. However, we can observe that for $\beta = 5$, FMDT achieves accuracy comparable to the other algorithms. On the other hand, FBDT and DDT are able to generate deeper trees. Note that deeper trees are more expressive and achieve higher accuracy on the training set, but they can be also affected by higher probability of over-training. However, FBDT tends to be more tolerant to the over-training issue than DDT. In particular, unlike DDT, for $\beta = 15$ FBDT achieves comparable results on both training and test sets, with the only exception for POK.

For the sake of completeness, we mention that the classification rates of both FMDT and FBDT are also higher than the ones reported in [177]. In [177], the authors investigate several prototype reduction techniques on Apache Hadoop with the aim of improving the classification rates of the nearest neighbor classifier. The experimental results on three big datasets have proven that these methods are very competitive in reducing the computational cost and high storage requirements of the nearest neighbor classifier, improving its classification performance. Due to the limited number of datasets adopted by the authors, we have not shown the results in Table 4.3, but however, we highlight that the average accuracy achieved by FMDT and FBDT in the common datasets is higher than the one obtained by the algorithm proposed in [177].

Table 4.5 shows the main characteristics of the partitions obtained by applying the fuzzy partitioning approach. In particular, the table reports the average number ($\overline{NFS}$) of fuzzy sets determined for the continuous attributes, the number of fuzzy sets for the attributes with the lowest (min_{NFS}) and highest (max_{NFS}) number of fuzzy sets, and the number of attributes DA discarded by the fuzzy partitioning process. Obviously, for POK, which is characterized by only categorical attributes, fuzzy partitioning is not performed.

Table 4.5: Complexities of Fuzzy Partitioning for both FMDT and FBDT.

	FMDT				**FBDT**			
Dataset	$\overline{NFS}$	min_{NFS}	max_{NFS}	DA	$\overline{NFS}$	min_{NFS}	max_{NFS}	DA
ECO	36.625	35	41	0	180.05	91	257	0
EME	36.875	35	42	0	176.225	98	245	0
HIG	8.229	3	32	6	10.136	3	42	6
KDD99_2	2.654	3	15	4	9.315	3	31	0
KDD99_5	3.3	3	15	4	15.131	3	42	0
KDD99	3.269	3	15	4	14.962	3	41	0
SUS	13.989	5	25	3	18.9	5	45	3

As shown in Table 4.5, for ECO, EME, HIG and SUS, fuzzy partitioning generates a high number of fuzzy sets, making the partitions hardly interpretable. For instance, ECO and EME are characterized, on average, by 180.05 and 176.225 triangular fuzzy sets per attribute. To limit the number of fuzzy set, a possible solution is to increment

the value of ϕ as exploited for FMDT. On the other hand, the parameter can affect the number of attribute discarded from the fuzzy partitioning. For instance, contrary to FBDT, for KDD99_2, KDD99_5 and KDD99, the algorithm removes 4 attributes that will be not employed by the FMDT.

Table 4.6 summarizes the execution times (in seconds) spent by each algorithm. For all algorithms we show the execution time of the tree learning processes (*Learning*), and for FMDT and FBDT, we report also the execution time performed by the fuzzy partitioning (*FP*) and the overall execution time (*Tot*). Here, the datasets have been split into a number of chunks equal to the number of cores available in the cluster, so that each core processes more or less the same number of instances.

Table 4.6: The execution time (in seconds) for FMDT, FBDT and DDT.

		FMDT			**FBDT**			**DDT**
Dataset	β	FP	$Learning$	Tot	FP	$Learning$	Tot	$Learning$
	5	28	364	392	29	64	93	11
ECO	10	-	-	-	29	215	244	13
	15	-	-	-	29	691	720	16
	5	23	349	372	24	42	66	11
EME	10	-	-	-	24	138	162	13
	15	-	-	-	24	579	603	17
	5	180	182	362	180	131	311	130
HIG	10	-	-	-	180	224	404	132
	15	-	-	-	180	424	604	149
	5	15	17	32	21	26	47	15
KDD99_2	10	-	-	-	21	49	70	16
	15	-	-	-	21	68	89	17
	5	16	24	40	30	41	71	17
KDD99_5	10	-	-	-	30	67	97	20
	15	-	-	-	30	86	116	21
	5	16	22	38	46	41	87	17
KDD99	10	-	-	-	46	67	113	19
	15	-	-	-	46	78	124	20
	5	-	3	3	-	4	4	4
POK	10	-	-	-	-	6	6	6
	15	-	-	-	-	11	11	11
	5	122	86	208	126	22	148	47
SUS	10	-	-	-	126	66	192	49
	15	-	-	-	126	130	255	54

As shown in Table 4.6, DDT is much faster than the other comparison algorithms: in general, the execution time is one order of magnitude lower than DFDT. Such result is mainly due to two different factors. First of all, the total execution time of FMDT and FBDT is affected by the fuzzy partitioning process. Such process is not performed by DDT, speeding up the execution time of the overall algorithm. Second, the amount of information managed by the FDT learning is higher than the one managed by the DDT learning. Indeed, since each point $x_{f,n} \in U_f$ belongs to two fuzzy sets, space complexity of FDT learning step is, in the worst case, twice than the one of DDT. The overall execution time of FBDT is comparable with the one obtained by FMDT. In particular, as shown in

Table 4.5, although FBDT employs a lower number of nodes than FMDT, it evaluates different binary splits for each attribute. However, the choice of the best split is bounded by the number of fuzzy sets defined on the attribute, which is significantly lower than the number of instances. On the other hand, FMDT can perform only one split for each attribute for a given node, thus speeding up the computation of the splitting procedure.

4.3.2 Scalability analysis

In this section, we investigate the scalability of the proposed approaches by employing an increasing number of CUs. To this aim, we measure the values assumed by the *speedup* σ that represents the main metrics used in parallel computing. According to the speedup definition, the efficiency of a program using multiple CUs is calculated comparing the execution time of the parallel implementation against the corresponding sequential version. Unfortunately, due to the large size of the involved datasets, the sequential version of the overall algorithm would take an unreasonable amount of time. Thus, for the scalability analysis we refer to a run over Q^* identical CUs, with $Q^* > 1$. With this aim, we adopt the following slightly different definition for the speedup on n identical CUs:

$$\sigma_{Q^*}(n) = \frac{Q^* \cdot \tau(Q^*)}{\tau(n)} \quad (4.16)$$

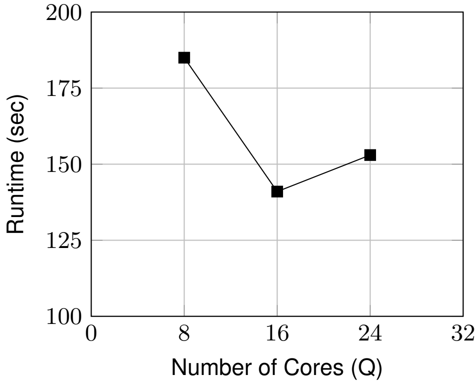
where $\tau(n)$ is the run-time using n CUs, and Q^* is the number of CUs used to run the reference execution, which lets us estimate a fictitious, ideal single-core run-time as $Q^* \cdot \tau(Q^*)$. Of course, $\sigma_{Q^*}(n)$ makes sense only for $n \geq Q^*$. Note that $\tau(Q^*)$ accounts also for the basic overhead due to the Apache Spark platform. Obviously, for $n > Q^*$ the speedup is expected to be sub-linear due to the increasing overhead from the Spark tasks, the behavior of the algorithm (considering also the granularity of the necessary sequential parts) and the contention for shared resources. In our tests, we assumed $Q^* = 8$ so as to have 1 working slave available in the cluster and thus accounting in σ_8 also for the basic overhead due to thread interference. Horizontal scalability has been studied by varying the number of switched-on CUs: we vary the number of slaves from 1 to 3, each with one executor with 8 cores. Considering the structure of our approach, we split the RDD into a number of partitions equal to the total number of cores available on the cluster.

Table 4.7 summarizes the results obtained on the Susy dataset by FBDT with $\beta = 15$. For the sake of brevity, we considered only one dataset and FBDT. However, similar results can be obtained on the other datasets and/or using FMDT.

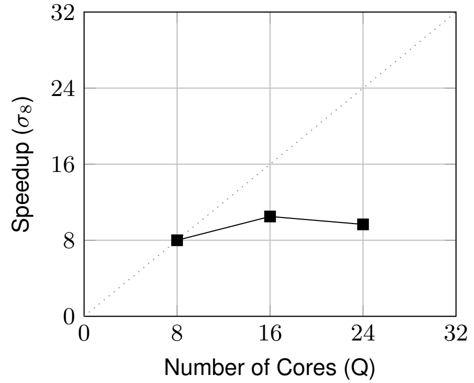
The actual speedup shows a different behavior depending on the algorithm. As regards Fuzzy Partitioning, σ_8 rapidly decreases and using 24 cores does not produce a real advantage; indeed the execution time with 24 cores is higher than the one obtained by using 16 cores. The result is mainly affected by two factors. First, the number of bins, namely $\Omega = V \cdot \gamma$, used to split the domain of each attribute is equal to 8,000, 16,000 and 24,000 for 8, 16 and 24 cores, respectively. Thus, in case of 24 cores, the amount of information handled by the algorithm is higher than the one handled for the other experiments,

Table 4.7: Run-time, speedup (σ_8), and utilization ($\sigma_8(Q)/Q$) of both Fuzzy Partitioning and FBDT Learning processes for the Susy dataset.

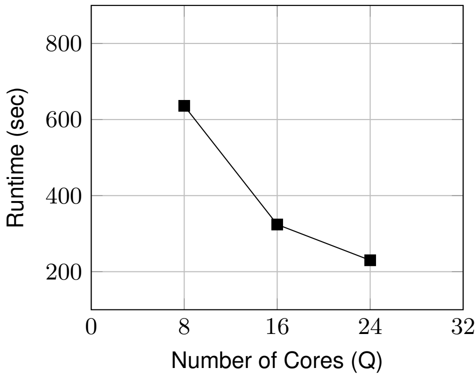
	Fuzzy Partitioning			Learning		
# Cores	**Time (s)**	$\sigma_8(Q)$	$\sigma_8(Q)/Q$	**Time (s)**	$\sigma_8(Q)$	$\sigma_8(Q)/Q$
8	185	8	1.00	636	8	1.00
16	141	10.50	0.66	324	15.70	0.98
24	153	9.67	0.40	230	22.12	0.92



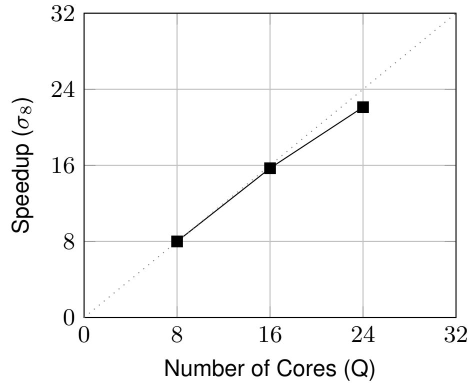
(a) Runtime of Fuzzy Partitioning.



(b) Speedup of Fuzzy Partitioning.



(c) Runtime of FBDT Learning.



(d) Speedup of FBDT Learning.

Figure 4.7: Speedup of Fuzzy Partitioning (a) and FBDT Learning (b) on the overall Susy dataset, varying the number of cores.

affecting the overall execution time. Second, the fuzzy partitioning of each continuous attribute is distributed among the cores available in the cluster so that each attribute is assigned to one core. Since Susy is characterized by 18 continuous attributes, each core processes approximately 3, 2 and 1 attributes in case of 8, 16 and 24 cores, respectively. However, as shown in Table 4.5, three attributes are discarded by the fuzzy partitioning process, thus for such attributes the overall process is performed in a few milliseconds (it requires exactly one scan for the exploration of candidate fuzzy partitions). Considering this result, the overall execution time can be roughly approximated with the same time required for 18-3=15 continuous attributes, thus each core processes approximately 2, 1 and 1 attributes in case of 8, 16 and 24 cores, respectively. The result highlights that, as regards the distribution of the computational flow, using a number of cores higher than 16 does not produce a real advantage and in such cases the execution time is only affected by the number of bins employed to explore the candidate fuzzy partitions.

As regard FBDT learning, σ_8 does not excessively diverge from the linear trend, i.e. the number of CUs: $\sigma_8(16)/16 = 0.98$ and $\sigma_8(24)/24 = 0.92$. The overhead is mainly due to higher number of executors handled by the Spark frameworks and the communication cost required to send the nodes that must be split from the master to the slaves.

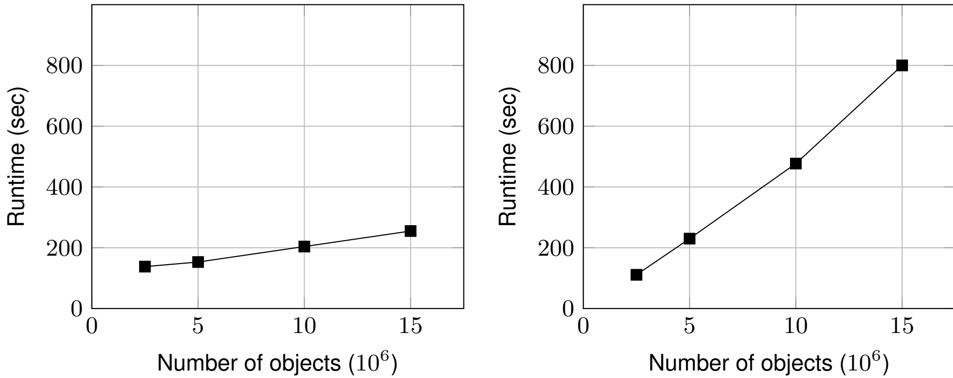
4.3.3 Dealing the dataset size

From a practical point of view, it is crucial to understand how the proposed algorithms behave as the size of the input dataset increases. To evaluate this aspect, we have performed several experiments using different dataset sizes. We have employed the Susy dataset and have used different percentages of this dataset. We indicate with the notation $Susy_x$ the dataset composed with x% of instances of the Susy dataset (the complete dataset is $Susy_{100}$). Moreover, we limit the experiments only to FBDT with $\beta = 15$ but similar considerations can be applied to FMDT.

Table 4.8 and Figure 4.8 show the run-time (in seconds) for building the tree (including the fuzzy partitioning), according to different dataset sizes. We report also the total number of instances N and the total number of instances in each chunk $N_v = N/V$. Like in the previous experiments, we distribute uniformly the entire dataset upon the number of available cores, i.e. $V = Q = 24$ in our tests. Note that for $Susy_{50}$, the average run-time of three different experiments executed over three distinct subsets of Susy (with instances randomly sampled) is reported.

Table 4.8: Run-time (in seconds) of FBDT on the Susy dataset, varying the dataset size.

Dataset			FBDT		
Size (%)	N	N_v	Fuzzy Partitioning	Learning	Tot
50 ($Susy_{50}$)	2,500,000	104,167	124	111	238
100 ($Susy_{100}$)	5,000,000	208,333	153	230	383
200 ($Susy_{200}$)	10,000,000	416,667	204	477	681
300 ($Susy_{300}$)	15,000,000	625,000	255	800	1055



(a) Runtime of Fuzzy Partitioning on different Susy dataset size

(b) Runtime of FBDT learning on different Susy dataset size

Figure 4.8: Runtime (in seconds) of Fuzzy Partitioning (a) and Learning (b) on the Susy dataset, varying the dataset size.

As shown in Figures 4.8a and 4.8b, the execution time of the two algorithms increases with different trends. However, the results are consistent with the time complexity analysis described in Section 4.2.3. As regards Fuzzy Partitioning, the computational cost is mainly driven by the number of bins Ω employed to explore the candidate fuzzy partitions. Since such value is constant in all tests, i.e. $\Omega = 24,000$, the execution time of the two reduce phases of fuzzy partitioning is more or less the same in all experiments. On the other hand, both map phases depend on the number of instances processed by each Map-Task. We recall that the first Map-Task performs a sorting of the instances for retrieving the equi-frequency bins and the second Map-Task computes for each bin the number of instances belonging to the different classes. Such operations are performed in $O(F \cdot N_v \cdot log(N_v))$ and $O(F \cdot N_v \cdot log(\Omega))$, respectively. However, considering the experiments and the number of instances involved, Ω and F are constants and $log(N_v)$ assumes more or less the same values (i.e. $log(N_v)$ ranges from about 5.02 to 5.8). Thus, we can expect that the run-time trend for both Map-Tasks is slightly higher than the linear one. These observations can be used to get a very rough estimation of the run-time expected for different dataset sizes. For instance, if adding 2,500,000 instances (from $Susy_{50}$ to $Susy_{100}$), the run-time increases of $153 - 129 = 24$ seconds, in the ideal case, we expect that adding 5,000,000 instances the execution time is slightly longer than twice. Thus we should obtain about $153 + 24 \times 2 = 201$ and $153 + 24 \times 4 = 249$ seconds for $Susy_{200}$ and $Susy_{300}$, respectively. As it can be noted, such values do not excessively differ from the measured ones. Of course, the actual run-times are necessarily higher due to the logarithmic factor $log(N_v)$ of the first Map-Task and the overheads for the sharing of memory resources.

As regards FBDT learning, we can perform the same observations exploited for Fuzzy Partitioning. In particular, as described in Section 4.2.3, time complexity of Reduce-Task

depends only on the number of splits, which have to be evaluated for computing the best splits among all attributes for the node, and is not affected by the number of instances. On the other hand, time complexity of Map-Task is equal to $O(N_v \cdot Y \cdot log(|D|))$. Since the number of nodes to split Y and the total number of fuzzy sets $|D|$ defined by Fuzzy Partitioning are more or less the same in all experiments, the overall run-time is mainly affected by N_v. Thus, increasing the number of instances, we expect that in the ideal case the execution time trend is linear, i.e. $111 \times 2 = 222$, $111 \times 4 = 444$ and $111 \times 6 = 666$ for $Susy_{100}$, $Susy_{200}$ and $Susy_{300}$, respectively. As it can be noted, such values do not excessively differ from the measured ones. Of course, the actual run-times are necessarily higher due to the overheads for the sharing of memory resources.

5

Multi-Objective Evolutionary Fuzzy System for Big Data

A number of methods have been proposed in the literature to generate and optimize the structure of a Fuzzy Rule-Based Classifier. While at the beginning these methods have mainly focused on optimizing the accuracy of the FRBCs, in the last years a particular attention has been also devoted to their interpretability. Indeed, one of the most appealing features of fuzzy rule-based classifiers is the capability of explaining how the conclusions are inferred. This feature is hard to preserve when fuzzy rules are extracted from a very large amount of data. Since accuracy and interpretability are conflicting objectives, the generation of the FRBC structure has been modeled as a multi-objective optimization problem. Multi-objective evolutionary algorithms (MOEAs) have been successfully employed to tackle this optimization problem with the main aim of generating sets of FRBCs characterized by different trade-offs between accuracy and interpretability [55, 62].

In this chapter, we propose a distributed implementation, denoted as DPAES-RCS, of PAES-RCS [14], a multi-objective evolutionary approach to learn concurrently the rule and data bases of FRBCs by maximizing accuracy and minimizing complexity. PAES-RCS has proven to be very efficient in obtaining satisfactory approximations of the Pareto front using a limited number of iterations [14]. This result has been obtained by learning the rule base through a rule and condition selection strategy, which selects a reduced number of rules from a heuristically generated set of candidate rules and a reduced number of conditions, for each selected rule, during the evolutionary process. We implemented DPAES-RCS on Apache Spark. We show the effectiveness of this implementation in terms of classification rate and scalability by using three real-world big datasets and comparing our results with the ones obtained by well-known state-of-art distributed algorithms. We highlight that the proposed approach allows handling big datasets even with modest hardware support.

The chapter is organized as follows. In Section 5.1, we introduce some preliminaries on FRBCs. Section 5.2 describes PAES-RCS in short and therefore DPAES-RCS and in Section 5.3, we illustrate the experimental results.

5.1 Distributed MOEA: state-of-the-art

Although different solutions for classification problems have been proposed for dealing with a huge amount of data [17, 38, 48, 175], only few works have integrated the fuzzy theory in their approaches [56, 119, 120]. All these works investigate the performance of classifiers only in terms of accuracy and scalability without considering interpretability. Indeed, to push up the accuracy of the model, the classifiers employ a high number of rules making them not interpretable.

In the last decades, multi-objective evolutionary algorithms (MOEAs) have been successfully employed for generating sets of FRBCs characterized by different trade-offs between accuracy and interpretability [55, 62]. However, since the computation of the accuracy of each solution generated in the evolutionary process requires the scan of the overall training set, when dealing with big data, the application of MOEA-based approaches to the FRBC generation is very critical. The solutions proposed so far in the literature have mainly focused on reducing the number of instances in the training set [10], by adopting some instance selection method, and on adopting techniques for speeding-up the convergence of the MOEA [14]. The use of the overall training set remains however a critical aspect when executing the MOEA on a unique machine, due mainly to storage and computational issues. Thus, the natural solution is to adopt a distributed approach on a computer cluster. Actually, in the last years, researchers have proposed several solutions to parallelize and distribute evolutionary algorithms [5, 15, 73, 96, 179] by investigating different models [15, 73], such as master-slaves, island, cellular, hierarchy, pool, coevolution and multi-agent models.

In a recent survey [73], authors highlight some different research hot-spots of distributed evolutionary algorithms and review several algorithms by classifying them according to the parallelism level, the adopted model and the infrastructure employed in their implementation (MPI, grid computing, P2P network, cloud computing and MapReduce, GPU and CUDA). In the following, we will briefly describe some of the approaches that are closer to the distributed implementation of the multi-objective evolutionary fuzzy system discussed in this chapter.

Exploiting the MapReduce paradigm, McNabb et al. [133] have proposed a particle swarm optimization able to scale up until 256 processors. Tagawa et al. [172] have investigated a concurrent differential evolution by testing the strategy on a multi-core processor. Chao et al. [94] have proposed MRPGA, an ad-hoc extension of MapReduce on a .NET-based enterprise grid system for parallelizing and therefore speeding up the computation of genetic algorithms. The attention on the MapReduce paradigm has been further pushed forward in 2007, with the first release of Apache Hadoop. Due to its simplicity and capability of handling very large datasets by scaling computational flow up to thousands of machines, researchers have implemented several distributed evolutionary algorithms on Hadoop not only for reducing the execution time of the algorithms, but also for investigating the opportunity of mining knowledge from big data. Thus, distributed implementations on Hadoop of genetic algorithms [69, 117, 174, 181], ant colony optimiza-

tion [198] and differential evolution [211] have been proposed and applied to different domains such as undersampling for imbalanced big data classification [174] and combinatorial optimization problems [198], achieving good performance in terms of scalability and proving the effectiveness of Hadoop in dealing with big data [117]. On the other hand, as shown in [211], the extra costs of the Hadoop distributed file system I/O operations and of the system bookkeeping overhead significantly reduce the benefits of parallelism. Hence, as we will discuss in Section 2.1, different new data processing environments that implement the concept of in-memory cluster computing should be employed. In [153], the authors have proposed a pairwise test generation based on parallel genetic algorithm and Apache Spark. The experimental study has however focused on a comparison with different algorithms without investigating scalability performance in terms of speedup.

5.2 The Proposed Algorithm

In this section, we first introduce the PAES-RCS algorithm proposed in [14]. Then, we describe in detail its distributed implementation DPAES-RCS on the Apache Spark framework.

5.2.1 PAES-RCS

The PAES-RCS algorithm generates a set of FRBCs with different trade-offs between accuracy and complexity by selecting rules and conditions from a set of candidate rules, and concurrently learning membership function parameters of the fuzzy sets used in the conditions of the rules. This objective is achieved by adopting a chromosome C composed of two parts (C_{RB}, C_{DB}), which define the RB and the membership function parameters of the input variables, respectively. We apply both crossover and mutation operators to each part of the chromosome independently. The set of candidate rules is extracted from a decision tree obtained by applying the well-known C4.5 algorithm to the training set.

Before applying the C4.5 algorithm, each continuous input variable X_f is transformed into a categorical and ordered variable by using a fuzzy uniform partition P_f of T_f triangular fuzzy sets: each category is the linguistic value corresponding to a fuzzy set in P_f. The category associated with each continuous value is determined by the index of the fuzzy set of the partition P_f to which the value belongs at maximum grade; in case of tie, we choose randomly.

Let $RB_{C4.5}$ and $M_{C4.5}$ be the RB generated by applying the C4.5 algorithm to the data set and the number of rules of this RB, respectively. Especially when dealing with large and high dimensional datasets, the C4.5 algorithm could generate RBs composed of a high number of rules. For this reason, in order to generate compact and interpretable RBs, we allow that the RBs of the solutions generated by PAES-RCS contain at most M_{MAX} rules. This value allows us to achieve a reasonable accuracy maintaining the complexity at an adequate level. Obviously, if the number of rules extracted from the

decision tree is lower than M_{MAX}, M_{MAX} is set to the actual number of rules extracted by the C4.5 algorithm.

The C_{RB} part of the chromosome is a vector of M_{MAX} pairs $\mathbf{p}_m = (k_m, \mathbf{v}_m)$, where $k_m \in [0, \ldots, M_{C4.5}]$ identifies the index of the rule in $RB_{C4.5}$ selected for the current RB and $\mathbf{v}_m = [v_{m,1}, \ldots, v_{m,F}]$ is a binary vector which indicates, for each condition in the rule, if the condition is present or corresponds to a "don't care". In particular, if $k_m = 0$, the mth rule is not included in the RB. Thus, we can generate RBs with a lower number of rules than M_{MAX}. Further, if $v_{m,f} = 0$, the fth condition of the mth rule is replaced by a "don't care" condition; otherwise it remains unchanged.

The C_{DB} part of the chromosome consists of F vectors of real numbers: the fth vector contains the $[b_{f,2}, ...b_{f,T_f-1}]$ cores which define the positions of the membership functions for the linguistic variable X_f. Indeed, we adopt triangular fuzzy sets $A_{f,j}$ defined by the tuple ($a_{f,j}$,$b_{f,j}$,$c_{f,j}$), where $a_{f,j}$ and $c_{f,j}$ correspond to the left and right extremes of the support of $A_{f,j}$, and $b_{f,j}$ to the core. Since we adopt strong fuzzy partitions with, for $j = 2, ..., T_f - 1$, $b_{f,j} = c_{f,j-1}$ and $b_{f,j} = a_{f,j+1}$, in order to define each fuzzy set of the partition it is sufficient to fix the positions of the cores $b_{f,j}$ throughout the universe U_f of the fth input variable. Since $b_{f,1}$ and b_{f,T_f} coincide with the extremes of the universe, the partition of each input variable X_f is completely defined by $T_f - 2$ parameters.

We apply the two-point crossover to the C_{RB} part and the BLX-α crossover, with $\alpha = 0.5$, to the C_{DB} part. As regards the mutation, for the C_{RB} part, we use two well-known operators, namely, random mutation [83] and flip-flop mutation [195]. Random mutation is also applied to the C_{DB} part.

As multi-objective evolutionary algorithm we use the (2+2)M-PAES that has been successfully employed in our previous works [10, 11, 12]. Unlike classical (2+2)PAES, in (2+2)M-PAES, current solutions are randomly extracted at each iteration rather than maintained until they are not replaced by solutions with particular characteristics.

5.2.2 The Distributed Approach

With the aim of dealing with big data, we propose DPAES-RCS, a distributed implementation of the PAES-RCS algorithm [14] on the Spark framework. DPAES-RCS consists of two main phases, which have been carefully designed for exploiting the potentialities of the distributed approach. In particular, the first phase, named *distributed candidate rules generation*, generates the candidate rule base that is employed by the distributed MOEA in the second phase, named *distributed evolutionary optimization*.

Let V be the number of *chunks* used for partitioning the training set and Q the number of Computing Units (CUs) available in the cluster. Each chunk $chunk_v$, with $v \in [1..V]$, contains a subset of $T = |N/V|$ instances $(\mathbf{x}_t^v, y_t^v)$ of the training set and feds only one *task*. On the other hand, a CU can process several tasks.

As shown in Figure 5.1, the *distributed candidate rules generation* consists of three steps. First, each continuous input variable X_f is discretized by using a uniform fuzzy partition with $T_f = 5$ fuzzy sets. For each $chunk_v$, we associate each pattern $x_{t,f}^v$ in $chunk_v$

with a categorical value corresponding to the label of the fuzzy set with the highest membership degree. The discretization is applied in parallel on each chunk of the RDD. Then, a distributed C4.5 algorithm is executed on the discretized training set. Finally, $RB_{C4.5}$ is extracted from the decision tree.

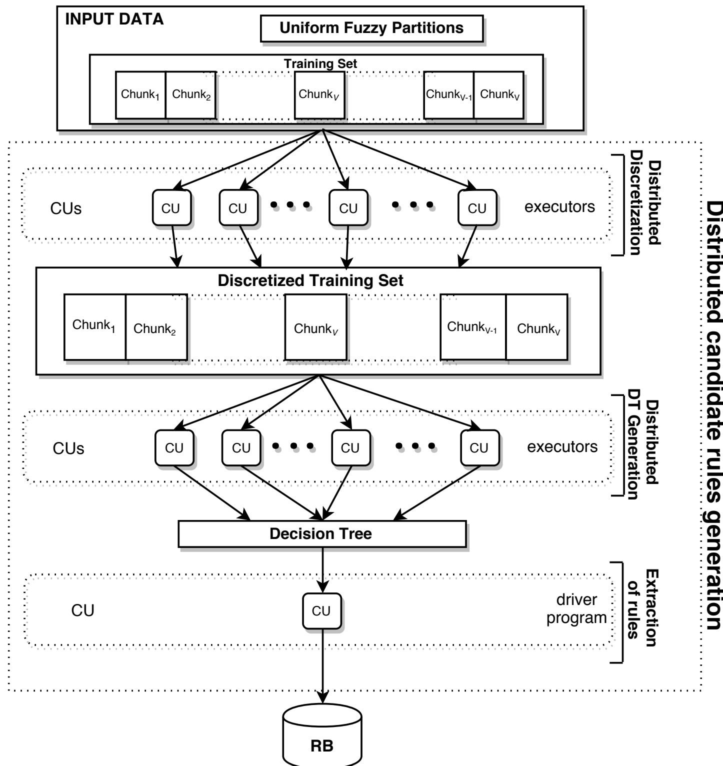


Figure 5.1: The distributed candidate rules generation phase.

As regards the distributed C4.5 algorithm for big datasets, we have modified the decision tree (DT) implementation[1] provided by MLlib [140]. In particular, similar to the CART algorithm [24], DT performs a recursive binary partitioning of the feature space. Each partition is chosen greedily by selecting the best split from a set of possible splits based on the information gain computed as difference between the parent node impurity and the weighted sum of the child node impurities. We have modified the DT implementation so as to manage categorical features as in the original C4.5. Thus, the arcs coming from a node labeled with a feature are labeled with each of the possible categorical values of the feature. If the node is associated with a discretized continuous input variable X_f, then

[1] For a complete description of this implementation, please refer to `https://spark.apache.org/docs/1.4.1/mllib-decision-tree.html`

it will have T_f arcs. Node impurity is computed by using entropy. Unlike classical C4.5 algorithm [155], no pruning step is performed.

Once the decision tree has been generated, $RB_{C4.5}$ is extracted from the tree. Then, two solutions are generated by using M_{MAX} rules randomly selected from $RB_{C4.5}$ and added to the 2+2MPAES archive for starting the execution of the *distributed evolutionary optimization* phase as shown in Figure 5.2.

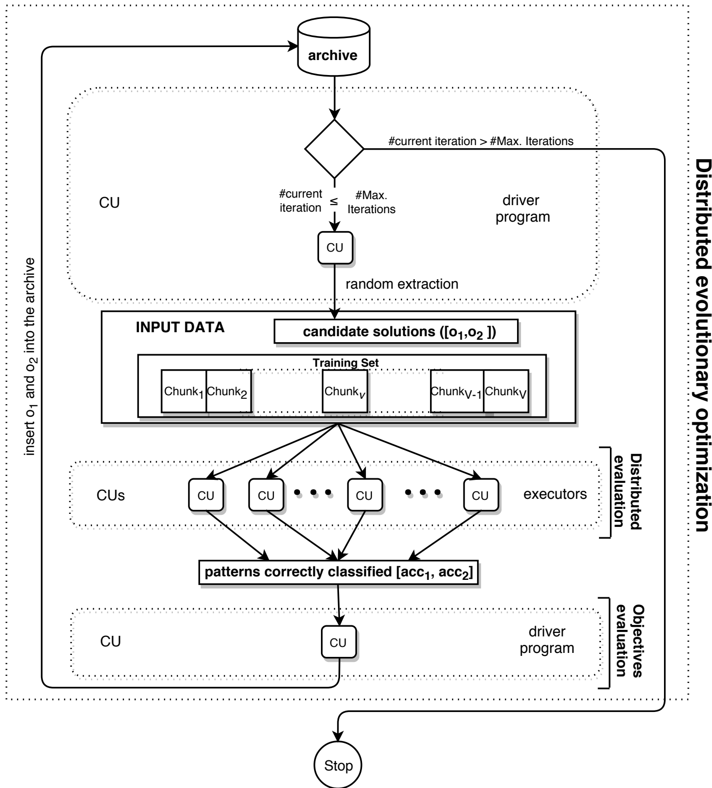


Figure 5.2: The distributed evolutionary optimization phase.

In particular, we distribute the computation of the classification rate at each iteration of the multi-objective evolutionary algorithm by adopting a master-slave model. Indeed, this computation is the most time-consuming part of the algorithm since it requires to scan the overall training set. More in detail, the *driver program* generates sequentially two new candidate offspring solutions, o_1 and o_2, by applying the genetic operators to two solutions, s_1 and s_2, randomly extracted from the archive. Each candidate solution is then evaluated by scanning the overall dataset: for each $chunk_v$, both o_1 and o_2 classify all the $\mathbf{x}_t^v$ instances belonging to the chunk.

Let acc_1 and acc_2 be the counters associated with o_1 and o_2, respectively. For each solution, if the pattern $\mathbf{x}_t^v$ is correctly labeled with y_t^v, the corresponding counter is incremented by 1. The counters have been implemented as *accumulators*, which are efficiently handled by Spark. After the RDD scan, the *driver program* computes the classification rate by dividing the values of the two accumulators by the number of instances in the training set. Further, the driver program calculates the total number of conditions that compose the antecedents of the rules in the RB, denoted as total rule length TRL. Finally, o_1 and o_2 are added to the archive only if they are dominated by no solution contained in the archive; possible solutions in the archive dominated by the candidate solutions are removed. Typically, the size of the archive is fixed at the beginning of the execution of the (2+2)M-PAES. In this case, when the archive is full and a new solution z has to be added to the archive, if z dominates no solution in the archive, then we insert z into the archive and remove the solution (possibly z itself) that belongs to the region with the highest crowding degree. If the region contains more than one solution, then, the solution to be removed is randomly chosen. The overall process is iteratively executed until the maximum number of fitness evaluations is achieved.

Note that the *discretization*, the *decision tree generation* and the *accuracy evaluation* procedures are distributed and executed concurrently on each of the V chunks of the RDD. Obviously, only Q tasks can be executed in parallel. Thus, if $V \leq Q$, then all tasks can be run simultaneously and the global runtime practically corresponds to the longest of the task runtimes. In case $V > Q$, only Q tasks can be executed in parallel and the rest ($V - Q$) tasks are queued, waiting for being executed as soon as one of the running Q task completes. Thus, in the ideal case where the execution time is the same for all tasks, each CU executes at most $\lceil \frac{V}{Q} \rceil$ tasks. In our experiments we have set $V = Q$.

5.3 Experimental Study

We tested our method on ten real-word big datasets extracted from the UCI repository[2]. As shown in Table 5.1, the datasets are characterized by different numbers of instances (up to 11 millions), attributes (from 10 to 54) and classes (from 2 to 23).

For each dataset, we performed a five-fold cross-validation with one trial for each fold. We implemented our algorithm by using Apache Spark 1.4.1 as distributed processing framework and we performed all the experiments using a small computer cluster. All the nodes are connected by a Gigabit Ethernet (1 Gbps) and run Ubuntu 12.04. The master node has a 4-core CPU (Intel Core i5 CPU 750 x 2.67 GHz), 8 GB of RAM and a 500GB Hard Drive and it is in charge to run the driver program. Each slave node is equipped by a 4-core CPU with Hyperthreading (Intel Core i7-2600K CPU x 3.40 GHz), 16GB of RAM and 1 TB Hard Drive. The training sets are stored in the Hadoop Distributed File System.

[2] https://archive.ics.uci.edu/ml/datasets.html

Table 5.1: Datasets used in the experiments.

Datasets			
Name	# Instances	# Attributes	# Classes
Covertype 2	581,012	54	2
Covertype 7	581,012	54	7
eCO	4,178,504	16	10
eME	4,178,504	16	10
Kddcup 2	4,856,151	41	2
Kddcup 5	4,898,431	41	5
Kddcup 23	4,898,431	41	23
Higgs	11,000,000	28	2
PokerHand	1,025,010	10	10
Susy	5,000,000	18	2

5.3.1 Performance of DPAES-RCS: accuracy and complexity

Table 5.2 reports the parameters used in the experiments. In order to determine the number of fitness evaluations we have exploited the work reported in [14], where we have proved that 50,000 fitness evaluations guarantee the same performance as 1 million evaluations. We observe that each iteration of the (2+2)M-PAES requires two fitness evaluations. Thus, the maximum number of iterations (# Max. Iterations) executed by (2+2)M-PAES is 25,000.

Since the C4.5 algorithm generates too many rules for Higgs and Susy datasets, we have tried different values for the parameter “minimum number of instances per leaf” in order to limit the number of candidate rules. Table 5.3 shows, for each dataset, the setting used for the C4.5 algorithm, the average number of rules generated and the average number of features selected at the end of the execution.

Table 5.2: Values of the parameters used in the experiments for DPAES-RCS.

# Fitness Evaluations	Maximum number of fitness evaluations	50,000
AS	(2+2)M-PAES archive size	64
T_f	Number of fuzzy sets in each variable X_f	5
M_{MAX}	Maximum number of rules in an RB	min(100, M_{C45})
$P_{C_{RB}}$	Probability crossover operator for C_{RB}	0.1
$P_{C_{DB}}$	Probability crossover operator for C_{DB}	0.5
$P_{M_{RB1}}$	Probability first mutation operator for C_{RB}	0.1
$P_{M_{RB2}}$	Probability second mutation operator for C_{RB}	0.7
$P_{M_{DB}}$	Probability mutation operator for C_{DB}	0.6

As in [11, 12], the evaluation of the solutions generated by DPAES-RCS is based on the analysis of three representative solutions of the Pareto front approximation: the most accurate (denoted as FIRST), the last accurate (denoted as LAST) and the median between the FIRST and the LAST (denoted as MEDIAN) solution. In practice, for each of the five folds, we compute the Pareto front approximation and sort the solutions in

Table 5.3: Values of the parameters used for the C4.5 and average number of rules and features in the rule bases extracted from the generated trees.

Dataset	min # instances per leaf	$\overline{Rules}$	$\overline{Features}$
Covertype 2	1	290.4	12.0
Covertype 7	1	18,176.20	53.04
eCO	1	591.4	12.0
eME	1	1,244	16.0
Higgs	1100	5,270.6	26.4
Kddcup 2	1	494.6	26.4
Kddcup 5	1	1,311	33.4
Kddcup 23	1	1,342.4	33.8
PokerHand	1	323,428.2	10.0
Susy	500	2,286	18.0

each approximation for increasing accuracy. Then, for each approximation, we select the first (the most accurate), the median and the last (the last accurate) solution (this is the reason why we denote the points as FIRST, MEDIAN and LAST).

Table 5.4 shows, for each dataset and for each solution, the average classification rate on both training (CR_{Train}) and test (CR_{Test}) sets, the average TRL, the average number NR of rules and the average number NA of attributes used in the rule base. Moreover, in Figure 5.3 we plot the mean values of Classification Rate and the Total Rule Length for the FIRST, MEDIAN and LAST solutions for all the datasets, on both training and test set, respectively.

Table 5.4: Average results obtained by the FIRST, MEDIAN and LAST solutions generated by DPAES-RCS

	FIRST					**MEDIAN**					**LAST**				
Dataset	CR_{Train}	CR_{Test}	TRL	NR	NA	CR_{Train}	CR_{Test}	TRL	NR	NA	CR_{Train}	CR_{Test}	TRL	NR	NA
Covertype 2	75.753	75.732	74.4	33.6	9.0	74.968	74.909	38.7	21.7	7.7	72.708	72.681	10.0	9.2	4.2
Covertype 7	72.383	72.374	145.0	36.2	32.0	71.940	71.924	84.2	29.4	25.9	57.921	57.907	58.2	28.0	24.0
eCO	77.133	77.115	168.4	54.0	12.0	74.995	74.984	117.7	45.4	11.8	56.228	56.244	54.4	35.2	11.4
eME	80.600	80.570	187.4	58.6	15.8	78.221	78.201	112.0	48.1	15.4	61.407	61.391	75.2	44.6	14.8
Higgs	65.008	64.998	125.2	30.2	19.0	64.389	64.370	78.7	25.8	15.2	59.825	59.849	48.6	23.2	14.4
Kddcup 2	99.948	99.947	35.4	21.8	12.6	99.933	99.934	19.5	13.2	8.8	98.508	98.514	8.2	8.0	5.4
Kddcup 5	99.740	99.734	80	34.8	17.8	99.717	99.711	50.4	26.5	14.1	82.920	82.925	30.2	23.4	13.2
Kddcup 23	99.802	99.803	77.8	33.2	20.0	99.735	99.735	42.2	23.5	15.7	63.384	63.374	23.6	20.0	11.2
PokerHand	60.233	60.221	113.2	50.0	5.0	58.423	58.430	68.1	35.2	5.0	48.772	48.749	34.2	25.4	5.0
Susy	78.123	78.11	80.4	28.0	13.6	77.658	77.659	45.6	19.9	12.1	68.131	68.128	22.0	15.0	9.6

Comparing the results obtained on the training and test sets in all datasets, we can state that our approach is not affected from the overtraining problem. Indeed, the values of the CR_{Train} for the FIRST, MEDIAN and LAST solutions are approximately equal to the ones of CR_{Test}.

To better investigate the goodness of our results, we compare the performance of DPAES-RCS with the ones achieved by the DT implementation available in MLlib [140].

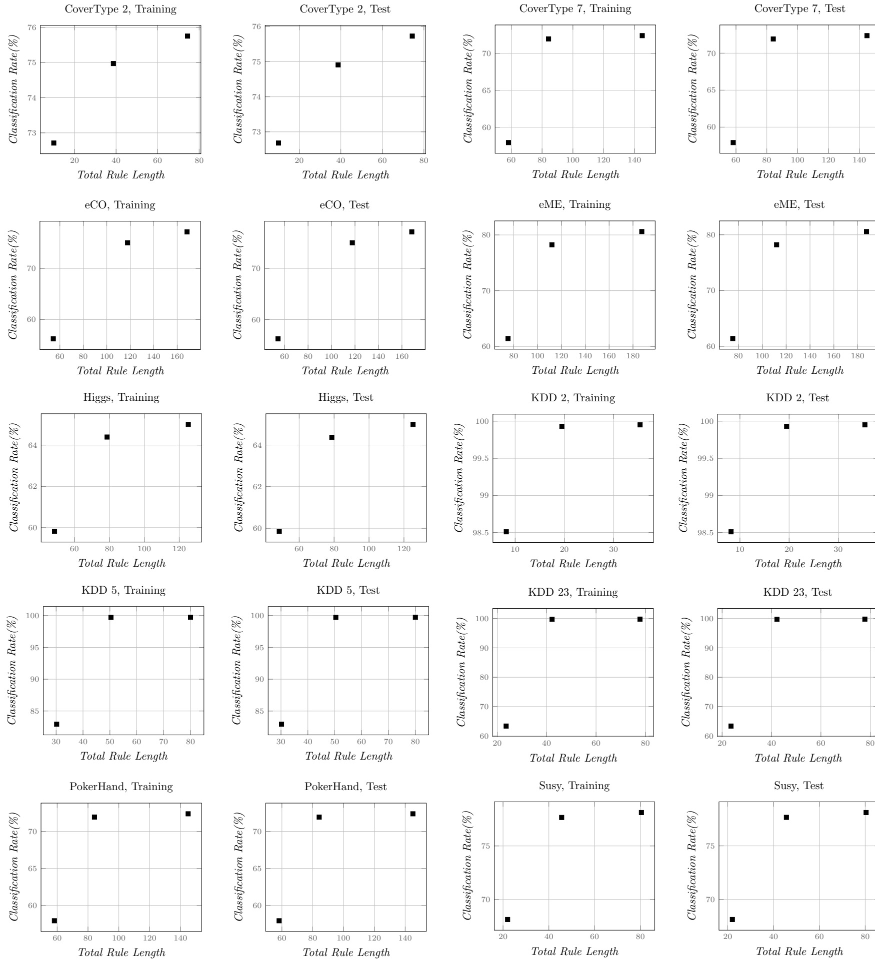


Figure 5.3: Average Pareto fronts obtained from each dataset.

As described in Section 5.2.2, this implementation performs a recursive binary partitioning of the feature space similar to the classical CART algorithm. To deal with big data, we follow the guidelines provided by MLlib: we set the *maximum depth* of the tree to 5 and the *number of bins* used to discretize continuous features to 32.

Table 5.5 shows the results achieved by the two approaches. The results have been obtained by employing the same folds for all the algorithms. For DPAES-RCS, we report only the results achieved by the FIRST solution. As regards the complexity of the decision-tree, we compute the total number of nodes (NN) and rules (NR), generated from DT. We observe that NR coincides with the total number of leaves of the model.

We highlight that we have not taken into consideration the number of antecedents because the ones exploited in the solutions of DPAES-RCS are different from the ones gen-

erated from the DT. Indeed, since DT performs a binary split at each node, it generates antecedents that are represented as intervals rather than single values as in DPAES-RCS. Obviously, the rules generated are more complex and hardly interpretable and a comparison considering TRL is not meaningful.

Table 5.5: Results of the application of DPAES-RCS and of the Decision Tree implemented in MLlib.

	DPAES-RCS				**Decision Tree**			
Datasets	CR_{Train}	CR_{Test}	TRL	NR	CR_{Train}	CR_{Test}	NN	NR
Covertype 2	**75.753**	**75.732**	74.4	33.6	73.890	73.810	62.2	**31.6**
Covertype 7	**72.383**	**72.374**	145.0	36.2	69.880	69.990	63.0	**32.0**
eCO	77.134	77.115	168.4	54.0	**77.907**	**77.895**	63.0	**32.0**
eME	**80.600**	**80.570**	187.4	58.6	78.085	78.048	62.2	**31.6**
Kddcup 2	99.948	99.947	35.4	**21.8**	**99.950**	**99.950**	43.0	22.0
Kddcup 5	**99.740**	**99.734**	80.0	34.8	99.693	99.694	58.2	**29.6**
Kddcup 23	**99.802**	**99.803**	77.8	33.2	99.730	99.730	48.2	**24.6**
Higgs	65.008	64.998	125.2	**30.2**	**66.338**	**66.336**	63.0	32.0
PokerHand	**60.233**	**60.221**	113.2	50.0	54.708	54.696	63.0	**32.0**
Susy	**78.123**	**78.110**	80.4	**28.0**	77.016	76.965	63.0	32.0

The analysis of Table 5.5 shows that DPAES-RCS outperforms DT in terms of accuracy. As regards the complexity, the number of rules of the FRBCs generated by DPAES-RCS is comparable with DT for the Covertype and Susy datasets, while it is higher for the eME dataset. We have to consider however that the rules extracted from the tree generated by DT are different from the rules generated by DPAES-RCS, since they have conditions with not only one linguistic value but with an “or” of linguistic values.

To statistically compare the three approaches, for both algorithms we generate a distribution consisting of the mean values of the accuracy of solutions on the test set by using all the datasets. Then, we apply a non-parametric test, namely Wilcoxon signed-rank test for pairwise comparison of two sample means [166]. Table 5.6 shows the result of Wilcoxon test. Here $R+$ and $R-$ represent the ranks corresponding to the DPAES-RCS and DT, respectively. We observe that the p-value is lower than the level of significance $\alpha = 0.10$. Thus, the null hypothesis is rejected.

Table 5.6: Results of the Wilcoxon signed-rank test on the CRs obtained on the test sets by DPAES-RCS and DT

	R+	R-	Hypothesis (α = 0.10)	p-value
DPAES-RCS vs DT	44	11	Rejected	0.0831

We would like to point out that the best average classification rate obtained on the Susy dataset by the K-NN classifier with prototype reduction proposed in [175] for dealing with big data classification was 72.82%. The *reduction rate*[3] is equal to 97.769%. The best classification rates obtained on the common datasets by MRAC+ (Table 3.3) and DAC-FFP (Table 3.18) are slightly higher than the ones achieved by DPAES-RCS, with the only exceptions for Kddcup 2 and Kddcup 23. Thus, both MRAC+ and DAC-FFP outperform DPAES-RCS in terms of accuracy but they use a larger number of rules, as shown in Table 3.5 and Table 3.19, respectively.

We can conclude that DPAES-RCS generates both accurate and interpretable FRBCs by employing a low number of fuzzy rules even with very large datasets.

5.3.2 Scalability analysis

In order to evaluate the scalability of the proposed approach, we use as metric the speedup, which is commonly used in parallel computing. As stated by the speedup definition, the efficiency of a program, which uses multiple CUs, can be calculated comparing the execution time of the parallel implementation against the corresponding sequential version. Unfortunately, due to the large size of the dataset, the sequential implementation of the algorithm is impracticable because it would take an unreasonable amount of time. To overcome this drawback, we take as reference a run over Q identical cores, with $Q > 1$. We redefine the speedup formula on n CUs as $\sigma_{Q^*}(n) = Q^* \cdot \tau(Q^*)/\tau(n)$, where $\tau(n)$ is the program runtime using n computing units and Q^* is the number of computing units used to run the reference execution. In our tests, we have assumed $Q^* = 8$. According to the scalability experiments, for avoiding unbalanced loads, we perform several executions by varying the number of computing units, keeping the same number of running cores per node. Obviously, $Q^*(n)$ makes sense only for $n \geq Q^*$, where the speedup is expected to be sub-linear due to the overhead from the Spark procedures and the contention of shared resources among cores. Indeed, we vary the number of slave nodes from 1 to 3, each of these with 8 cores and one executor. Moreover, considering the structure of DPAES-RCS, we split the RDD into a number of partitions equal to the total number of cores available on the slaves.

Table 5.7 summarizes the results. Figures 5.4 and 5.5 show the speedup and the execution time, respectively, obtained on the Covertype 2 dataset.

As shown in Figures 5.4 and 5.5, the speedup does not excessively differ from a linear trend. The overhead is mainly due to the time required to send the solutions (C_{RB}, C_{DB}) from the master to each slave node.

[3] The measure has been computed as $1 - size(RS)/size(TR)$, where $size(RS)$ is the total number of prototypes of the NN classifier and $size(TR)$ is the total number of objects in the training set.

Table 5.7: Speedup of the PAES-RCS algorithm in classification.

# Cores	Time (s)	Speedup	$\sigma_8(Q)/Q$ (Utilization)
8	13,297.264	8.0	1.0
16	6,901.795	15.4131	0.9633
24	4,858.456	21.8955	0.9123

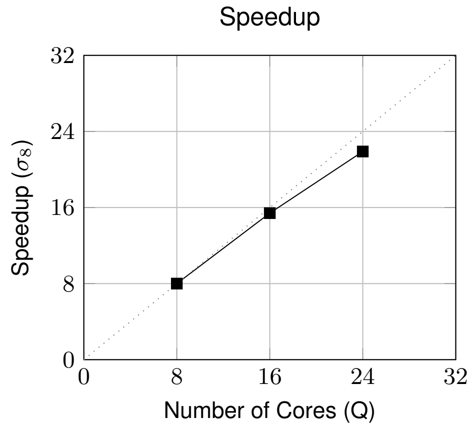


Figure 5.4: Speedup of DPAES-RCS varying the number of cores

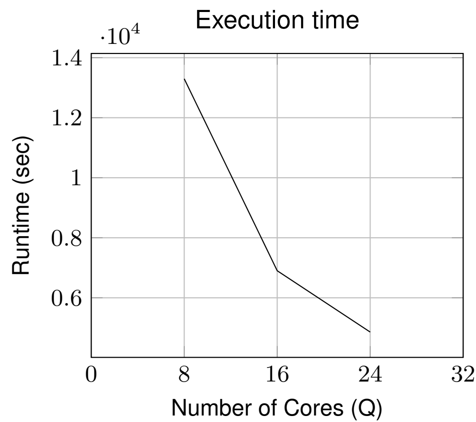


Figure 5.5: Execution time of DPAES-RCS varying the number of cores

6

Conclusions

In this Ph.D thesis, we have proposed different solutions shaped according to the MapReduce programming model for handling Big Data and generating accurate and interpretable classifier. In particular, we have focused on associative classification and decision trees, integrating our solutions with fuzzy set theory as well. Furthermore, to increase the interpretability of classifiers, we have proposed a distributed multi-objective evolutionary approach to learn concurrently the rule and data bases of FRBCs by maximizing accuracy and minimizing complexity.

As regards associative classification, we have proposed MRAC, a MapReduce Associative Classifier based on frequent pattern mining. MRAC first extracts the frequent items; then, it generates the most significant classification association rules from them by exploiting a modified parallel version of the well-known FP-Growth algorithm; finally, it prunes noisy and redundant rules by applying a distributed dataset coverage approach. Memory usage and time complexity have been shown for each phase of the learning process. We have also discussed MRAC+, a faster version of MRAC, which does not employ the dataset coverage, exploiting instead the best rule inference mechanism. Both MRAC and MRAC+ are able to process millions of data for learning the classification association rules. Experimental results performed on seven big datasets show that MRAC+ and MRAC are able to achieve speedup and scalability close to the ideal ones.These results have been compared against with achieved on the same hardware platform by two state-of-the-art distributed classification algorithms, namely the Random Forest (RF) in Mahout over Hadoop, and the Decision Tree (DT) in MLlib over Spark; in the comparison, accuracy, complexity, and runtime have been taken into account. We have concluded that, considering accuracy, MRAC+ achieves higher classification rates than MRAC, outperforming DT, and with results that closely compare to those obtained by RF. From the runtime perspective, MRAC+ is faster than MRAC, but both their implementations over Hadoop are much slower than the Spark implementation of DT (not surprisingly, considering the typical efficiency gap between the Spark and Hadoop platforms).

We have also proposed a new efficient fuzzy association rule-based classification scheme based on a fuzzy version of the well-known FP-Growth algorithm, denoted as

AC-FFP. Moreover, we have presented a MapReduce version of AC-FFP, denoted as DAC-FFP, able to deal with big data. The development of the classification schemes has required to introduce a number of purposely-defined strategies for: (i) appropriately generating the fuzzy partitions, (ii) extending the FP-Growth algorithm to the fuzzy context, (iii) selecting the most accurate and non-redundant fuzzy rules and (iv) performing the classification of unlabeled patterns. More in detail, for AC-FFP, the strong fuzzy partitions have been generated by appropriately defining fuzzy sets on crisp partitions obtained by applying a classical discretization algorithm for continuous attributes. On the other hand, DAC-FFP employs a distributed fuzzy partitioning based on fuzzy entropy for generating Ruspini fuzzy partitions for each continuous feature. The FP-Growth has been extended to the fuzzy context by adopting proper definitions of fuzzy support and confidence. Further, unlike previous extensions proposed in the literature, which reduce the complexity by considering only the most frequent fuzzy value for each attribute in the generation of the fuzzy rules, we have considered all the frequent fuzzy sets for each attribute. For the third strategy, we have defined and applied three different types of rule pruning. Finally, for the classification of unlabeled patterns, we have proposed an adjustment of the classical reasoning method with the aim of balancing the importance of both general and specific rules. As regards the accuracy of the generated fuzzy associative classifiers, AC-FFP has been tested on seventeen small classification benchmarks. We have compared the results achieved by our fuzzy model with the ones achieved by a well-known non-fuzzy associative classification model, namely the CMAR algorithm, and by two state-of-the-art algorithms for generating fuzzy rule-based association classifiers, namely FARC-HD and D-MOFARC. By performing non-parametric statistical tests, we have highlighted that the proposed approach outperforms the CMAR algorithm, in terms of classification accuracy on the test set, and achieves accuracies similar to FARC-HD and D-MOFARC. However, we have to highlight that FARC-HD and D-MOFARC employ a fuzzy adaptation of the Apriori algorithm for mining the fuzzy rules and an evolutionary post-processing for pruning these rules and optimizing the fuzzy partitions. On the other hand, DAC-FFP has been tested on six real-world big datasets. Scalability analysis shows that DAC-FFP is able to achieve speedup close to the ideal achievable targets. As regards the accuracy, the results show that DAC-FFP outperforms MRAC in all the employed datasets and achieves accuracies similar to or better than MRAC+. Moreover, DAC-FFP employs a lower number of rules than both MRAC+ and MRAC, but on the other hand is much more slower than the comparison algorithms. In particular, DAC-FFP employs an additional MapReduce pruning step than MRAC, increasing the exectuion time of the overall learning process.

As regards tree-based classification, we have proposed a distributed fuzzy decision tree learning scheme shaped according to the MapReduce programming paradigm. As well as DAC-FFP, the overall algorithm first employs the distributed fuzzy partitions for generating Ruspini fuzzy partition for each continuous attribute. Then, the fuzzy partitions are used as input to the distributed MapReduce-based tree learning algorithm. In particular, we have adopted two different versions of the learning algorithm based on

multi-way and binary splits, namely FMDT and FBDT, respectively. Both the versions employ the information gain computed in terms of fuzzy entropy for selecting the attribute to be adopted at each decision node. In the experimental study, we have compared our proposals with the decision tree available in MLlib on eight real-world big datasets. Both, FMDT and FBDT outperform decision tree in most of the employed datasets. Further, although binary splits are able to generate deeper trees than multi-way splits, FBDT tends to be more tolerant to overtraining issues than decision trees. Comparing trees with the same depth, FMDT outperforms binary splits approaches due to the higher number of nodes that can be generated at each decision node, affecting both complexity of the tree and execution time.

As regards multi-objective evolutionary algorithm (MOEA), we have proposed a distributed, scalable and effective implementation of PAES-RCS, namely DPAES-RCS, which learns concurrently the rule and data bases of fuzzy rule-based classifiers by maximizing accuracy and minimizing complexity. The overall algorithm consists of two main phases. In the first phase, we have employed a distributed step for generating candidate rules. These rules are then randomly extracted to build the starting rule base that is employed by the distributed MOEA in the second phase. In particular, we have adopted a master-slave model approach deployed on Apache Spark. Thanks to the in-memory computation, Apache Spark is more suitable for iterative and online applications than Apache Hadoop. In the experimental study, we have compared our proposal with the decision tree available in MLlib on ten real-world big datasets, pointing out that our approach generates accurate classifiers with very low complexity. Moreover, by performing non parametric statistical tests, we have highlighted that the proposed approach outperforms the distributed decision tree implemented in MLlib, in terms of classification accuracy on the test set. Further, scalability analysis shows that the algorithm achieves speedup close to the ideal achievable targets.

References

1. Neda Abdelhamid, Aladdin Ayesh, Fadi Thabtah, Samad Ahmadi, and Wael Hadi. MAC: A Multiclass Associative Classification Algorithm. *Journal of Information & Knowledge Management*, 11(02), 2012.
2. Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining Association Rules between Sets of Items in Large Databases. *SIGMOD Record*, 22(2):207–216, June 1993.
3. Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.
4. Moh'd Iqbal AL Ajlouni, Wa'el Hadi, and Jaber Alwedyan. Detecting Phishing Websites Using Associative Classification. *European Journal of Business and Management*, 5(15):36–40, 2013.
5. Enrique Alba and José M Troya. A survey of parallel distributed genetic algorithms. *Complexity*, 4(4):31–52, 1999.
6. J. Alcala-Fdez, R. Alcala, and F. Herrera. A fuzzy association rule-based classification model for high-dimensional problems with genetic rule selection and lateral tuning. *IEEE Transactions on Fuzzy Systems*, 19(5):857–872, 2011.
7. J. Alcalá-Fdez, L. Sánchez, S. García, M.J. Jesus, S. Ventura, J.M. Garrell, J. Otero, C. Romero, J. Bacardit, V.M. Rivas, J.C. Fernández, and F. Herrera. KEEL: a software tool to assess evolutionary algorithms for data mining problems. *Soft Computing*, 13(3):307–318, 2009.
8. Nasullah Khalid Alham, Maozhen Li, Yang Liu, and Suhel Hammoud. A MapReduce-based distributed SVM algorithm for automatic image annotation. *Computers & Mathematics with Applications*, 62(7):2801–2811, 2011.
9. Xavier Amatriain. Mining Large Streams of User Data for Personalized Recommendations. *ACM SIGKDD Explorations Newsletter*, 14(2):37–48, 2013.
10. M. Antonelli, P. Ducange, and F. Marcelloni. Genetic Training Instance Selection in Multi-Objective Evolutionary Fuzzy Systems: A Co-evolutionary Approach. *IEEE Transactions on Fuzzy Systems*, 20(2):276–290, 2012.
11. Michela Antonelli, Pietro Ducange, Beatrice Lazzerini, and Francesco Marcelloni. Learning concurrently data and rule bases of mamdani fuzzy rule-based systems by exploiting a novel interpretability index. *Soft Computing*, 15(10):1981–1998, 2011.
12. Michela Antonelli, Pietro Ducange, Beatrice Lazzerini, and Francesco Marcelloni. Learning knowledge bases of multi-objective evolutionary fuzzy systems by simultaneously optimizing accuracy, complexity and partition integrity. *Soft Computing*, 15(12):2335–2354, 2011.

13. Michela Antonelli, Pietro Ducange, and Francesco Marcelloni. An efficient multi-objective evolutionary fuzzy system for regression problems. *International Journal of Approximate Reasoning*, 54(9):1434 – 1451, 2013.
14. Michela Antonelli, Pietro Ducange, and Francesco Marcelloni. A fast and efficient multi-objective evolutionary learning scheme for fuzzy rule-based classifiers. *Information Sciences*, 283:36–54, 2014.
15. Jaume Bacardit and Xavier Llorà. Large-scale data mining using genetics-based machine learning. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 3(1):37–61, 2013.
16. Elena Baralis and Paolo Garza. A lazy approach to pruning classification rules. In *Proceedings of 2002 IEEE International Conference on Data Mining*, pages 35–42, 2002.
17. Alessio Bechini, Francesco Marcelloni, and Armando Segatori. A MapReduce solution for associative classification of big data. *Information Sciences*, 332:33–55, 2016.
18. Fernando Berzal, Juan-Carlos Cubero, Nicolás Marın, and Daniel Sánchez. Building multi-way decision trees with numerical attributes. *Information Sciences*, 165(1):73–90, 2004.
19. Neha Bharill and Aruna Tiwari. Handling Big Data with Fuzzy based Classification Approach. In *Advance Trends in Soft Computing*, pages 219–227. Springer, 2014.
20. Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. Moa: Massive online analysis. *The Journal of Machine Learning Research*, 11:1601–1604, 2010.
21. Danah Boyd and Kate Crawford. Critical Questions for Big Data. *Information, Communication & Society*, 15(5):662–679, 2012.
22. Xavier Boyen and Louis Wehenkel. Automatic induction of fuzzy decision trees and its application to power system security assessment. *Fuzzy Sets and Systems*, 102(1):3–19, 1999.
23. L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.
24. Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and regression trees*. CRC press, 1984.
25. Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. *Proceedings of the VLDB Endowment*, 3(1-2):285–296, 2010.
26. Gregory Buehrer, Srinivasan Parthasarathy, Shirish Tatikonda, Tahsin Kurc, and Joel Saltz. Toward Terabyte Pattern Mining: An Architecture-conscious Solution. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 2–12, New York, NY, USA, 2007. ACM.
27. G. Caruana, Maozhen Li, and Man Qi. A MapReduce based parallel SVM for large scale spam filtering. In *Fuzzy Systems and Knowledge Discovery (FSKD), 2011 Eighth International Conference on*, volume 4, pages 2659–2662, July 2011.
28. B Chandra and P Paul Varghese. Fuzzy SLIQ decision tree algorithm. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 38(5):1294–1301, 2008.
29. Guoqing Chen, Hongyan Liu, Lan Yu, Qiang Wei, and Xing Zhang. A new approach to classification based on association rule mining. *Decision Support Systems*, 42(2):674–689, 2006.
30. Hsinchun Chen, Roger H. L. Chiang, and Veda C. Storey. Business Intelligence and Analytics: From Big Data to Big Impact. *MIS Q.*, 36(4):1165–1188, December 2012.
31. Wen-Chin Chen, Chiun-Chieh Hsu, and Yu-Chun Chu. Increasing the effectiveness of associative classification in terms of class imbalance by using a novel pruning algorithm. *Expert Systems with Applications*, 39(17):12841–12850, 2012.
32. Wen-Chin Chen, Chiun-Chieh Hsu, and Jing-Ning Hsu. Adjusting and generalizing CBA algorithm to handling class imbalance. *Expert Systems with Applications*, 39(5):5907–5919, 2012.
33. Yi-lai Chen, Tao Wang, Ben-sheng Wang, and Zhou-jun Li. A Survey of Fuzzy Decision Tree Classifier. *Fuzzy Information and Engineering*, 1(2):149–159, 2009.
34. Zuoliang Chen and Guoqing Chen. Building an associative classifier based on fuzzy association rules. *International Journal of Computational Intelligence Systems*, 1(3):262–273, 2008.

35. X.-Q. Cheng, X.-L. Jin, Y.-Z. Wang, J.-F. Guo, T.-Y. Zhang, and G.-J. Li. Survey on big data system and analytic technology. *Ruan Jian Xue Bao/Journal of Software*, 25(9):1889–1908, 2014.
36. Zheru Chi, Hong Yan, and Tuân Pham. *Fuzzy Algorithms: With Applications to Image Processing and Pattern Recognition*, volume 10. World Scientific, 1996.
37. Chin-Tzong Pang Chien-Hua Wang, Wei-Hsuan Lee. Applying Fuzzy FP-Growth to Mine Fuzzy Association Rules. *International Journal of Computer, Electrical, Automation, Control and Information Engineering*, 4(5):788–794, 2010.
38. Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-Reduce for Machine Learning on Multicore. In Bernhard Schölkopf, John C. Platt, and Thomas Hoffmanan, editors, *Advances in Neural Information Processing Systems 19, Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems Vancouver, British Columbia, Canada, December 4-7, 2006*, pages 281–288. MIT Press, 2006.
39. Ellis J. Clarke and Bruce A. Barton. Entropy and MDL discretization of continuous variables for Bayesian belief networks. *International Journal of Intelligent Systems*, 15(1):61–92, 2000.
40. Oscar Cordón, María José del Jesus, and Francisco Herrera. A proposal on reasoning methods in fuzzy rule-based classification systems. *International Journal of Approximate Reasoning*, 20(1):21–45, 1999.
41. Gianni Costa, Riccardo Ortale, and Ettore Ritacco. X-Class: Associative classification of XML documents by structure. *ACM Transactions on Information Systems*, 31(1):3:1–3:40, 2013.
42. GM Cramer, RA Ford, and RL Hall. Estimation of toxic hazard—a decision tree approach. *Food and cosmetics toxicology*, 16(3):255–276, 1976.
43. Wei Dai and Wei Ji. A MapReduce Implementation of C4.5 Decision Tree Algorithm. *International Journal of Database Theory and Application*, 7(1):49–60, 2014.
44. Adriano Donato De Matteis, Francesco Marcelloni, and Armando Segatori. A new approach to fuzzy random forest generation. In *Fuzzy Systems (FUZZ-IEEE), 2015 IEEE International Conference on*, pages 1–8. IEEE, 2015.
45. Andrea De Mauro, Marco Greco, and Michele Grimaldi. What is big data? A consensual definition and a review of key research topics. *AIP Conference Proceedings*, 1644(1):97–104, 2015.
46. Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, January 2008.
47. Jeffrey Dean and Sanjay Ghemawat. MapReduce: A Flexible Data Processing Tool. *Commun. ACM*, 53(1):72–77, January 2010.
48. Sara del Río, Victoria Løpez, Josè Manuel Benítez, and Francisco Herrera. On the use of MapReduce for imbalanced big data using Random Forest. *Information Sciences*, 285:112–137, 2014.
49. Joaquín Derrac, Salvador García, Daniel Molina, and Francisco Herrera. A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms. *Swarm and Evolutionary Computation*, 1(1):3 – 18, 2011.
50. Ruisheng Diao, Kai Sun, Vijay Vittal, Robert J O'Keefe, Michael R Richardson, Navin Bhatt, Dwayne Stradford, and Sanjoy K Sarawgi. Decision tree-based online voltage security assessment using pmu measurements. *Power Systems, IEEE Transactions on*, 24(2):832–839, 2009.
51. James Dougherty, Ron Kohavi, and Mehran Sahami. Supervised and Unsupervised Discretization of Continuous Features. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 194–202. Morgan Kaufmann, 1995.
52. Apache Drill. `http://drill.apache.org`, Accessed: May 2016.
53. Dryad. `http://research.microsoft.com/en-us/collaboration/tools/dryad.aspx/`, Accessed: May 2016.

54. Sumeet Dua, Harpreet Singh, and H.W. Thompson. Associative classification of mammograms using weighted rules. *Expert Systems with Applications*, 36(5):9250–9259, 2009.
55. Pietro Ducange and Francesco Marcelloni. Multi-objective evolutionary fuzzy systems. In *Proceedings of the 9th international conference on Fuzzy Logic and Applications*, pages 83–90. Springer-Verlag, 2011.
56. Pietro Ducange, Francesco Marcelloni, and Armando Segatori. A MapReduce-based fuzzy associative classifier for big data. In *Proceedings of the 2015 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, pages 1–8, 2015.
57. Richard O Duda, Peter E Hart, and David G Stork. *Pattern classification*. John Wiley & Sons, 2012.
58. Erin-Elizabeth Durham, Xiaxia Yu, Robert W Harrison, et al. FDT 2.0: Improving scalability of the fuzzy decision tree induction tool-integrating database storage. In *Computational Intelligence in Healthcare and e-health (CICARE), 2014 IEEE Symposium on*, pages 187–190. IEEE, 2014.
59. Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: A Runtime for Iterative MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 810–818. ACM, 2010.
60. Jaliya Ekanayake, Shrideep Pallickara, and Geoffrey Fox. MapReduce for Data Intensive Scientific Analyses. In *eScience, 2008. eScience'08. IEEE Fourth International Conference on*, pages 277–284. IEEE, 2008.
61. Usama M. Fayyad and Keki B. Irani. Multi-Interval Discretization of Continuous-Valued Attributes for Classification Learning. In *Proceedings of IJCAI*, pages 1022–1029, 1993.
62. M. Fazzolari, R. Alcalá, Y. Nojima, H. Ishibuchi, and F. Herrera. A Review of the Application of Multi-Objective Evolutionary Fuzzy Systems: Current Status and Further Directions. *IEEE Transactions on Fuzzy Systems*, 21(1):45–65, 2013.
63. Michela Fazzolari, Rafael Alcalá, and Francisco Herrera. A multi-objective evolutionary method for learning granularities based on fuzzy discretization to improve the accuracy-complexity trade-off of fuzzy rule-based classification systems: D-MOFARC algorithm. *Applied Soft Computing*, 24:470–481, 2014.
64. Apache Flink. `https://flink.apache.org/`. Accessed: May 2016.
65. Apache FlinkML. `https://ci.apache.org/projects/flink/flink-docs-master/libs/ml/`. Accessed: May 2016.
66. Apache Flink Streaming. `https://flink.apache.org/news/2015/02/09/streaming-example.html`. Accessed: May 2016.
67. María José Gacto, Rafael Alcalá, and Francisco Herrera. Interpretability of linguistic fuzzy rule-based systems: An overview of interpretability measures. *Information Sciences*, 181(20):4340–4360, 2011.
68. Sergio Garcia, Julián Luengo, José Antonio Sáez, Victor López, and Francisco Herrera. A survey of discretization techniques: taxonomy and empirical analysis in supervised learning. *Knowledge and Data Engineering, IEEE Transactions on*, 25(4):734–750, 2013.
69. Linda Di Geronimo, Filomena Ferrucci, Alfonso Murolo, and Federica Sarro. A Parallel Genetic Algorithm based on Hadoop MapReduce for the Automatic Generation of JUnit Test Suites. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, pages 785–793, 2012.
70. Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, October 2003.
71. Apache Giraph. `http://giraph.apache.org/`. Accessed: May 2016.
72. Thomas Goetz. *The decision tree: taking control of your health in the new era of personalized medicine*. Rodale, 2010.
73. Y.-J. Gong, W.-N. Chen, Z.-H. Zhan, J. Zhang, Y. Li, Q. Zhang, and J.-J. Li. Distributed evolutionary algorithms and their models: A survey of the state-of-the-art. *Applied Soft Computing Journal*, 34:286–300, 2015.

74. Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, 2012.
75. GraphLab. `https://dato.com/products/create/open_source.html`. Accessed: May 2016.
76. H2O. `http://h2o.ai/`. Accessed: May 2016.
77. Apache Hadoop. `http://hadoop.apache.org`, Accessed: May 2016.
78. Apache Hama. `https://hama.apache.org/`. Accessed: May 2016.
79. Jiawei Han, Micheline Kamber, and Jian Pei. *Data mining: concepts and techniques*. Elsevier, 2011.
80. Jiawei Han, Jian Pei, Yiwen Yin, and Runying Mao. Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach. *Data Min. Knowl. Discov.*, 8(1):53–87, 2004.
81. Trevor Hastie, Robert Tibshirani, Jerome Friedman, and James Franklin. The Elements of Statistical Learning: data mining, inference and prediction. *The Mathematical Intelligencer*, 27(2):83–85, 2005.
82. Qing He, Changying Du, Qun Wang, Fuzhen Zhuang, and Zhongzhi Shi. A parallel incremental extreme SVM classifier. *Neurocomputing*, 74(16):2532–2540, 2011.
83. Francisco Herrera, Manuel Lozano, and Jose L. Verdegay. Tackling real-coded genetic algorithms: Operators and tools for behavioural analysis. *Artificial intelligence review*, 12(4):265–319, 1998.
84. Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, volume 11, pages 22–22, 2011.
85. Tzung-Pei Hong, Yeong-Chyi Lee, and Min-Thai Wu. An effective parallel approach for genetic-fuzzy data mining. *Expert Systems with Applications*, 41(2):655 – 662, 2014.
86. H. Hu, Y. Wen, T.-S. Chua, and X. Li. Toward scalable systems for big data analytics: A technology tutorial. *IEEE Access*, 2:652–687, 2014.
87. Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, March 2007.
88. H. Ishibuchi and T. Nakashima. Effect of Rule Weights in Fuzzy Rule-Based Classification Systems. *IEEE Transactions on Fuzzy Systems*, 9(4):506–515, 2001.
89. H. Ishibuchi and T. Yamamoto. Rule Weight Specification in Fuzzy Rule-Based Classification Systems. *IEEE Transactions on Fuzzy Systems*, 13(4):428–435, 2005.
90. Hisao Ishibuchi, Satoshi Mihara, and Yusuke Nojima. Parallel Distributed Hybrid Fuzzy GBML Models With Rule Set Migration and Training Data Rotation. *IEEE Transactions on Fuzzy Systems*, 21(2):355–368, 2013.
91. Hisao Ishibuchi, Tomoharu Nakashima, and Manabu Nii. *Classification and Modeling with Linguistic Information Granules: Advanced Approaches to Linguistic Data Mining (Advanced Information Processing)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2004.
92. Cezary Z Janikow. A genetic algorithm method for optimizing fuzzy decision trees. *Information Sciences*, 89(3):275–296, 1996.
93. Cezary Z Janikow. Fuzzy Decision Trees: Issues and Methods. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 28(1):1–14, 1998.
94. Chao Jin, Christian Vecchiola, and Rajkumar Buyya. MRPGA: An extension of MapReduce for parallelizing genetic algorithms. In *Proceedings of the 4th IEEE International Conference on eScience, eScience 2008*, pages 214–221, 2008.
95. X. Jin, B.W. Wah, X. Cheng, and Y. Wang. Significance and Challenges of Big Data Research. *Big Data Research*, 2(2):59–64, 2015.

96. Fauzi Mohd Johar, Farah Ayuni Azmin, Mohamad Kadim Suaidi, Abdul Samad Shibghatullah, Badrul Hisham Ahmad, Siti Nadzirah Salleh, Mohamad Zoinol Abidin Abd Aziz, and M Md Shukor. A Review of Genetic Algorithms and Parallel Genetic Algorithms on Graphics Processing Unit (GPU). In *2013 IEEE International Conference on Control System, Computing and Engineering (ICCSCE)*, pages 264–269, 2013.
97. U. Kang, D.H. Chau, and C. Faloutsos. Pegasus: Mining billion-scale graphs in the cloud. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pages 5341–5344, March 2012.
98. Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. *Learning Spark: Lightning-Fast Big Data Analysis*. " O'Reilly Media, Inc.", 2015.
99. Hyunjoong Kim and Wei-Yin Loh. Classification trees with unbiased multiway splits. *Journal of the American Statistical Association*, 2011.
100. Sotiris Kotsiantis and Dimitris Kanellopoulos. Discretization techniques: A recent survey. *GESTS International Transactions on Computer Science and Engineering*, 32(1):47–58, 2006.
101. Sotiris B Kotsiantis, I Zaharakis, and P Pintelas. Supervised Machine Learning: A Review of Classification Techniques, 2007.
102. Tim Kraska. Finding the Needle in the Big Data Systems Haystack. *IEEE Internet Computing*, 17(1):84–86, 2013.
103. Lukasz A Kurgan and Krzysztof J Cios. Caim Discretization Algorithm. *Knowledge and Data Engineering, IEEE Transactions on*, 16(2):145–153, 2004.
104. Douglas Laney. 3D Data Management: Controlling Data Volume, Velocity, and Variety. Technical report, META Group, February 2001.
105. Bingguo Li, Xiaojun Chen, Mark Junjie Li, Joshua Zhexue Huang, and Shengzhong Feng. Scalable Random Forests for Massive Data. In *Advances in Knowledge Discovery and Data Mining*, pages 135–146. Springer, 2012.
106. Haoyuan Li, Yi Wang, Dong Zhang, Ming Zhang, and Edward Y. Chang. PFP: Parallel FP-Growth for Query Recommendation. In *Proceedings of the 2008 ACM Conference on Recommender Systems*, RecSys '08, pages 107–114, New York, NY, USA, 2008. ACM.
107. Lingjuan Li and Min Zhang. The Strategy of Mining Association Rule Based on Cloud Computing. In *Proceedings of the 2011 International Conference on Business Computing and Global Informatization, BCGIn 2011*, pages 475–478, July 2011.
108. Wenmin Li, Jiawei Han, and Jian Pei. CMAR: Accurate and Efficient Classification based on Multiple Class-Association Rules. In *Proceedings of IEEE International Conference on Data Mining 2001*, pages 369–376, 2001.
109. Chun-Wei Lin, Tzung-Pei Hong, and Wen-Hsiang Lu. Linguistic data mining with fuzzy FP-trees. *Expert Systems with Applications*, 37(6):4560 – 4567, 2010.
110. Jimmy Lin. MapReduce is good enough? If all you have is a hammer, throw away everything that's not a nail! *Big Data*, 1(1):28–37, 2013.
111. Jimmy Lin and Dmitriy Ryaboy. Scaling big data mining infrastructure: the twitter experience. *ACM SIGKDD Explorations Newsletter*, 14(2):6–19, 2013.
112. Ming-Yen Lin, Pei-Yu Lee, and Sue-Chen Hsueh. Apriori-based Frequent Itemset Mining Algorithms on MapReduce. In *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*, ICUIMC '12, pages 76:1–76:8, New York, NY, USA, 2012. ACM.
113. 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*, pages 80–86, 1998.
114. Bing Liu, Yiming Ma, and Ching-Kian Wong. Classification using association rules: Weaknesses and enhancements. In RobertL. Grossman, Chandrika Kamath, Philip Kegelmeyer, Vipin Kumar, and RajuR. Namburu, editors, *Data Mining for Scientific and Engineering Applications*, volume 2 of *Massive Computing*, pages 591–605. Springer US, 2001.

115. Li Liu, Eric Li, Yimin Zhang, and Zhizhong Tang. Optimization of Frequent Itemset Mining on Multiple-core Processor. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1275–1285. VLDB Endowment, 2007.
116. Xiaodong Liu, Xinghua Feng, and Witold Pedrycz. Extraction of fuzzy rules from fuzzy decision trees: An axiomatic fuzzy sets (AFS) approach. *Data & Knowledge Engineering*, 84:1–25, 2013.
117. Xavier Llora, Abhishek Verma, Roy H Campbell, and David E Goldberg. When huge is routine: scaling genetic algorithms and estimation of distribution algorithms via data-intensive computing. In *Parallel and Distributed Computational Intelligence*, pages 11–41. Springer, 2010.
118. Wei-Yin Loh and Nunta Vanichsetakul. Tree-structured classification via generalized discriminant analysis. *Journal of the American Statistical Association*, 83(403):715–725, 1988.
119. Victoria López, Sara del Río, José Manuel Benítez, and Francisco Herrera. On the use of MapReduce to build linguistic fuzzy rule based classification systems for big data. In *2014 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, pages 1905–1912, 2014.
120. Victoria López, Sara del Río, José Manuel Benítez, and Francisco Herrera. Cost-sensitive linguistic fuzzy rule based classification systems under the MapReduce framework for imbalanced big data. *Fuzzy Sets and Systems*, 258:5–38, 2015.
121. Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.
122. Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. GraphLab: A New Framework for Parallel Machine Learning. *CoRR*, abs/1006.4990, 2010.
123. Joel Pinho Lucas, Anne Laurent, Maria N Moreno, and Maguelonne Teisseire. A fuzzy associative classification approach for recommender systems. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 20(04):579–617, 2012.
124. S. Otto M. Snir. *MPI-The Complete Reference: The MPI Core*. MIT Press, 1998.
125. Yue Ma, Guoqing Chen, and Qiang Wei. A novel business analytics approach and case study–fuzzy associative classifier based on information gain and rule-covering. *Journal of Management Analytics*, 1(1):1–19, 2014.
126. Sam Madden. From Databases to Big Data. *IEEE Internet Computing*, 16(3):4–6, 2012.
127. Apache Mahout. `http://mahout.apache.org`, Accessed: May 2016.
128. Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.
129. Kulwinder Singh Mann and Navjot Kaur. Cloud-deployable health data mining using secured framework for Clinical decision support system. In *Computing and Communication (IEMCON), 2015 International Conference and Workshop on*, pages 1–6, Oct 2015.
130. Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, Alan J. Miller, and Michael Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 6(1):4–15, February 2002.
131. Vivien Marx. Biology: The big challenges of big data. *Nature*, 498(7453):255–260, 2013.
132. Nathan Marz and James Warren. *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co., 2015.
133. Andrew W McNabb, Christopher K Monson, and Kevin D Seppi. Parallel PSO Using MapReduce. In *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, pages 7–14. IEEE, 2007.
134. Swarup Medasani, Jaeseok Kim, and Raghu Krishnapuram. An overview of membership function generation techniques for pattern recognition. *International Journal of approximate reasoning*, 19(3):391–417, 1998.

135. Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive Analysis of Web-scale Datasets. *Proc. VLDB Endow.*, 3(1-2):330–339, September 2010.
136. Dimitris Meretakis and Beat Wuthrich. Extending Naive Bayes Classifiers Using Long Itemsets. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '99, pages 165–174, New York, NY, USA, 1999. ACM.
137. Apache Mesos. `http://mesos.apache.org/`. Accessed: May 2016.
138. Ambiga Dhiraj Michael Minelli, Michele Chambers. *Big data big analytics: emerging business intelligence and analytic trends for today's businesses*. John Wiley & Sons, 2013.
139. Debahuti Mishra, Shruti Mishra, Sandeep Kumar Satapathy, and Srikanta Patnaik. Genetic Algorithm based Fuzzy Frequent Pattern Mining from Gene Expression Data. In *Soft Computing Techniques in Vision Science*, pages 1–14. Springer, 2012.
140. Apache MLlib. `https://spark.apache.org/mllib/`, Accessed:February 2016.
141. MOA - Massive Online Analysis. `http://moa.cms.waikato.ac.nz/`. Accessed: May 2016.
142. Muriel J Montbriand. Decision tree model describing alternate health care choices made by oncology patients. *Cancer Nursing*, 18(2):117, 1995.
143. Derek G Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. Ciel: a universal execution engine for distributed data-flow computing. In *Proc. 8th ACM/USENIX Symposium on Networked Systems Design and Implementation*, pages 113–126, 2011.
144. AJ Myles and SD Brown. Induction of decision trees using fuzzy partitions. *Journal of chemometrics*, 17(10):531–536, 2003.
145. L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed Stream Computing Platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177, Dec 2010.
146. Loan T.T. Nguyen, Bay Vo, Tzung-Pei Hong, and Hoang Chi Thanh. CAR-miner: An efficient algorithm for mining class-association rules. *Expert Systems with Applications*, 40(6):2305–2311, 2013.
147. Albert Orriols-Puig, Francisco J Martínez-López, Jorge Casillas, and Nick Lee. Unsupervised KDD to creatively support managers' decision making with fuzzy association rules: A distribution channel application. *Industrial Marketing Management*, 42(4):532–543, 2013.
148. Sean Owen, Robin Anil, Ted Dunning, and Ellen Friedman. *Mahout in Action*. Manning Shelter Island, 2011.
149. Ferenc Peter Pach, Attila Gyenesei, and Janos Abonyi. Compact fuzzy association rule-based classifier. *Expert Systems with Applications*, 34(4):2406 – 2416, 2008.
150. Indranil Palit and Chandan K. Reddy. Scalable and Parallel Boosting with MapReduce. *Knowledge and Data Engineering, IEEE Transactions on*, 24(10):1904–1916, Oct 2012.
151. Bin Pei, Tingting Zhao, Suyun Zhao, and Hong Chen. Fuzzy Associative Classifier for Probabilistic Numerical Data. In *Foundations and Applications of Intelligent Systems*, pages 563–578. Springer, 2014.
152. Iko Pramudiono and Masaru Kitsuregawa. Parallel FP-Growth on PC Cluster. In *Advances in Knowledge Discovery and Data Mining*, volume 2637 of *Lecture Notes in Computer Science*, pages 467–473. Springer Berlin Heidelberg, 2003.
153. RZ Qi, ZJ Wang, and SY Li. Pairwise Test Generation Based on Parallel Genetic Algorithm with Spark. In *International Conference on Computer Information Systems and Industrial Applications*. Atlantis Press, 2015.
154. J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.
155. J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.
156. Anand Rajaraman, Jeffrey D Ullman, Jeffrey David Ullman, and Jeffrey David Ullman. *Mining of massive datasets*, volume 1. Cambridge University Press Cambridge, 2012.

157. Sergio Ramirez-Gallego, Salvador Garcia, Hector Mourino-Talin, and David Martinez-Rego. Distributed Entropy Minimization Discretizer for Big Data Analysis under Apache Spark. In *Trustcom/BigDataSE/ISPA, 2015 IEEE*, volume 2, pages 33–40. IEEE, 2015.
158. Sergio Ramírez-Gallego, Salvador García, Héctor Mouriño-Talín, David Martínez-Rego, Verónica Bolón-Canedo, Amparo Alonso-Betanzos, José Manuel Benítez, and Francisco Herrera. Data discretization: taxonomy and big data challenge. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2015.
159. Brian D. Ripley. *Pattern recognition and neural networks*. Cambridge university press, 1996.
160. Miguel Rodríguez, Diego M Escalante, and Antonio Peregrín. Efficient Distributed Genetic Algorithm for Rule Extraction. *Applied soft computing*, 11(1):733–743, 2011.
161. Ansel Y Rodríguez-González, José Fco Martínez-Trinidad, Jesús A Carrasco-Ochoa, and José Ruiz-Shulcloper. Mining frequent patterns and association rules using similarities. *Expert Systems with Applications*, 40(17):6823–6836, 2013.
162. Lior Rokach and Oded Maimon. *Data mining with decision trees: theory and applications*. World scientific, 2014.
163. Diego Sánchez-Moreno, Ana Belén Gil, and María N Moreno. TV-SeriesRec: A recommender system based on fuzzy associative classification and semantic information. In *Trends in Practical Applications of Agents and Multiagent Systems*, pages 201–208. Springer, 2013.
164. Tawny Schlieski and Brian David Johnson. Entertainment in the Age of Big Data. *Proceedings of the IEEE*, 100(Special Centennial Issue):1404–1408, 2012.
165. Sangwon Seo, Edward J Yoon, Jaehong Kim, Seongwook Jin, Jin-Soo Kim, and Seungryoul Maeng. HAMA: An Efficient Matrix Computation with the MapReduce Framework. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 721–726. IEEE, 2010.
166. David J Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. crc Press, 2003.
167. Pritpal Singh. Big Data Time Series Forecasting Model: A Fuzzy-Neuro Hybridize Approach. In *Computational Intelligence for Big Data Analysis*, pages 55–72. Springer, 2015.
168. Apache Spark. `https://spark.apache.org`, Accessed: May 2016.
169. Apache Spark Streaming. `http://spark.apache.org/streaming/`, Accessed: May 2016.
170. Apache Storm. `https://storm.apache.org`, Accessed: May 2016.
171. Yanmin Sun, Yang Wang, and Andrew K.C. Wong. Boosting an associative classifier. *Knowledge and Data Engineering, IEEE Transactions on*, 18(7):988–992, July 2006.
172. Kiyoharu Tagawa and Takashi Ishimizu. Concurrent differential evolution based on MapReduce. *International Journal of Computers*, 4(4):161–168, 2010.
173. F. Thabtah. A review of associative classification mining. *Knowledge Engineering Review*, 22(1):37–65, 2007.
174. I Triguero, M Galar, S Vluymans, C Cornelis, H Bustince, F Herrera, and Y Saeys. Evolutionary undersampling for imbalanced big data classification. In *Proceedings of IEEE Congress on Evolutionary Computation (CEC 2015)*, 2015.
175. I. Triguero, D. Peralta, J. Bacardit, S. García, and F. Herrera. MRPR: A MapReduce solution for prototype reduction in big data classification. *Neurocomputing*, 150(PA):331–345, 2015.
176. Isaac Triguero, Sara del Río, Victoria López, Jaume Bacardit, José M Benítez, and Francisco Herrera. ROSEFW-RF: The winner algorithm for the ECBDL'14 big data competition: An extremely imbalanced big data bioinformatics problem. *Knowledge-Based Systems*, 87:69–79, 2015.
177. Isaac Triguero, Daniel Peralta, Jaume Bacardit, Salvador García, and Francisco Herrera. MRPR: A MapReduce solution for prototype reduction in big data classification. *neurocomputing*, 150:331–345, 2015.
178. Twister: Iterative MapReduce. `http://iterativemapreduce.org/`. Accessed: May 2016.
179. AJ Umbarkar and MS Joshi. Review of parallel genetic algorithm based on computing paradigm and diversity in search space. *ICTACT Journal on Soft Computing*, 3:615–622, 2013.

180. Adriano Veloso, Wagner Meira Jr., and Mohammed J. Zaki. Lazy Associative Classification. In *Proceedings of the Sixth International Conference on Data Mining*, ICDM '06, pages 645–654, Washington, DC, USA, 2006. IEEE Computer Society.
181. Abhishek Verma, Xavier Llora, David E Goldberg, and Roy H Campbell. Scaling genetic algorithms using MapReduce. In *ISDA 2009 - 9th International Conference on Intelligent Systems Design and Applications*, pages 13–18, 2009.
182. Bay Vo, Tzung-Pei Hong, and Bac Le. A lattice-based approach for mining most generalization association rules. *Knowledge-Based Systems*, 45(0):20–30, 2013.
183. Vowpal wabbit. `http://hunch.net/~vw/`, Accessed: May 2016.
184. Dingxian Wang, Xiao Liu, and Mengdi Wang. A DT-SVM Strategy for Stock Futures Prediction with Big Data. In *Computational Science and Engineering (CSE), 2013 IEEE 16th International Conference on*, pages 1005–1012. IEEE, 2013.
185. Li-Xin Wang and Jerry M Mendel. Generating fuzzy rules by learning from examples. *Systems, Man and Cybernetics, IEEE Transactions on*, 22(6):1414–1427, 1992.
186. Ran Wang, Yu-Lin He, Chi-Yin Chow, Fang-Fang Ou, and Jian Zhang. Learning ELM-Tree from big data based on uncertainty reduction. *Fuzzy Sets and Systems*, 258:79–100, 2015.
187. Shanshan Wang, Junhai Zhai, Hong Zhu, and Xizhao Wang. Parallel Ordinal Decision Tree Algorithm and Its Implementation in Framework of MapReduce. In *Machine Learning and Cybernetics*, pages 241–251. Springer, 2014.
188. Xianchang Wang, Xiaodong Liu, Witold Pedrycz, and Lishi Zhang. Fuzzy rule based decision trees. *Pattern Recognition*, 48(1):50–59, 2015.
189. XiZhao Wang, Daniel S. Yeung, and Eric C. C. Tsang. A comparative study on heuristic algorithms for generating fuzzy decision trees. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 31(2):215–226, 2001.
190. R. Weber. Automatic knowledge acquisition for fuzzy control applications. In *Proc. International Symposium on Fuzzy Systems (Japan). July*, pages 9–12, 1992.
191. Rosina Weber. Fuzzy-id3: a class of methods for automatic knowledge acquisition. In *The second international conference on fuzzy logic and neural networks*, pages 265–268, 1992.
192. S. Wedyan. Review and Comparison of Associative Classification Data Mining Approaches. *International Journal of Computer, Electrical, Automation, Control and Information Engineering*, 8(1):34 – 45, 2014.
193. Louis Wehenkel and Mania Pavella. Decision tree approach to power systems security assessment. *International Journal of Electrical Power & Energy Systems*, 15(1):13–36, 1993.
194. Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc, 3rd edition edition, 2012.
195. Darrell Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.
196. Frank Wilcoxon. Individual Comparisons by Ranking Methods. *Biometrics Bulletin*, 1(6):pp. 80–83, 1945.
197. Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Series in Data Management Sys. Morgan Kaufmann, 3rd edition, June 2011.
198. Bihan Wu, Gang Wu, and Mengdong Yang. A MapReduce based Ant Colony Optimization approach to combinatorial optimization problems. In *The Eighth International Conference on Natural Computation (ICNC)*, pages 728–732, 2012.
199. Xindong Wu, Xingquan Zhu, Gong-Qing Wu, and Wei Ding. Data mining with big data. *Knowledge and Data Engineering, IEEE Transactions on*, 26(1):97–107, 2014.
200. J. Xin, Z. Wang, L. Qu, and G. Wang. Elastic extreme learning machine for big data classification. *Neurocomputing*, 149(PA):464–471, 2015.
201. Xiaoxin Yin and Jiawei Han. CPAR: Classification based on predictive association rules. In *Proceedings of the third SIAM international conference on data mining*, pages 331–335, 2003.
202. Yongwook Yoon and Gary G. Lee. Two scalable algorithms for associative text classification. *Information Processing and Management*, 49(2):484–496, 2013.

203. Yufei Yuan and Michael J Shaw. Induction of fuzzy decision trees. *Fuzzy Sets and systems*, 69(2):125–139, 1995.
204. H. Yuwen. Cost-sensitive incremental Classification under the MapReduce framework for Mining Imbalanced Massive Data Streams. *Journal of Discrete Mathematical Sciences and Cryptography*, 18:177–194, 2015.
205. Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2, 2012.
206. Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.
207. O.R. Zaiane, M. El-Hajj, and P. Lu. Fast parallel association rule mining without candidacy generation. In *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*, pages 665–668, 2001.
208. Mohsen Zeinalkhani and Mahdi Eftekhari. Fuzzy partitioning of continuous attributes through discretization methods to construct fuzzy decision tree classifiers. *Information Sciences*, 278:715–735, 2014.
209. Chi Zhang, Feifei Li, and Jeffrey Jestes. Efficient parallel kNN joins for large data in MapReduce. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 38–49, New York, NY, USA, 2012. ACM.
210. Yu Zheng, Like Liu, Longhao Wang, and Xing Xie. Learning transportation mode from raw gps data for geographic applications on the web. In *Proceedings of the 17th international conference on World Wide Web*, pages 247–256. ACM, 2008.
211. Chi Zhou. Fast parallelization of differential evolution algorithm using MapReduce. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 1113–1114, 2010.
212. Le Zhou, Zhiyong Zhong, Jin Chang, Junjie Li, J.Z. Huang, and Shengzhong Feng. Balanced parallel FP-Growth with MapReduce. In *Proc. of 2010 IEEE Youth Conference on Information Computing and Telecommunications (YC-ICT)*, pages 243–246, Nov 2010.

Acknowledgments

Foremost, my deepest and sincere gratitude is to my advisor Prof. Francesco Marcelloni, for his patient, continuous support and motivation during my studies and related research. I have been extremely lucky to have such advisor who gave me the freedom to explore on my own but at the same time helped me when my steps faltered. Despite the myriad of commitments, he has been always present and his expertise and knowledge have been a precious treasure for my research. I should like to take this opportunity to congratulate Prof. Marcelloni for his recent promotion to Full Professor. The extra hours spent for working on several fields such as research, projects, and teaching, and the careful attention to the details have been a symbol and guide to follow for me and my future career. Best wishes for continued success.

Moreover, I would like to acknowledge the other two professors Pietro Ducange and Alessio Bechini for their contribution, encouragement and practical advices in the most of my papers and projects. I consider myself very fortunate to have had the possibility to work with them. Besides the academic activities, I would like to express my congratulations to Pietro for the newborn daughter. I know that he will be a lovely dad and I truly wish all the best to him and his family.

During my doctoral, I spent about four months at the University of Alberta in Canada as Ph.D visiting student. They have been one of the most important experience of my doctorate and life as well. In that period, I had the occasion of collaborating with Prof. Witold Pedrycz who supervised my research while in Edmonton whose results are discussed in the fourth chapter. I want to thank Prof. Pedrycz. Further, in Canada I met very good people from all around the world and I want to say thank to all of them, especially to my friends Orion, Eliezyer and Riccardo, Tom and my roommates Richards, Spencer and Steve. It has been nice spent times with them, tasting different typical foreign dishes and beers and hiking in the Rocky Mountains.

I am also grateful to all my colleagues and especially to my labmates of Department of Information Engineering. It was nice alternating times of hard work with funny moments and jokes and I wish to all of them the best for their careers.

I would warmly thank all my friends of my home town, especially “Real Moda & Pelle” football team, Mario Emme, Manuel (Lele), Er Pera (Alessandro), Claudio (Er Drago/Satana), Daniele (Er Biondo), Pierpaolo (Billy), Guido and Maurizio (Er Banana). I shared beautiful and funny moments with them from the first years of my life and I established invaluable friendship despite the distance and the time.

During my last ten years in Pisa, I have met always wonderful people. Among them I would like to thank my football teams “I Passi”, “Tremanza” and “Le Fere”. It was an honor have played with them as well as the special dinners spent together. A grateful thanks goes to my friends that I met during my first years at the university, especially to Antonio (Gigione), Francesco, Alessandro (Loki) and Daniela, Daniele e Santina, Chiara and Anna, Andrea (Who?) and Viviana. They let me laugh everyday even when I had to face some unhappy moments.

Special thanks go to my best friend Marco. He has been always present and even if in the last year we are living in different countries, he has still supported, encouraged, entertained, and helped me every times. I consider myself really lucky to have met a guy like him and I’m sure that our “bromance” is strong enough to be stable despite distance and time.

A heartfelt thanks goes to Federica for being such a patient and comprehensive girlfriend even when I was only thinking about simulations and optimizations of code. Her smile and spontaneity have always made my days, especially in harsh times. She is definitively the best that could happen to me in the last years and I hope she will be on my side when I will face any kind of problems.

Last but not the least, I would like to thank my family: my beloved parents and my lovely sister (and his husband Vittorio) for supporting me financially and spiritually throughout my studies and my life as well. I would not be able to become who I am now without their precious advices, constant comprehension, and boundless love and kindness. I hope one day to be a good example for my family as well as you have been for me.

Finally, I owe my gratitude to all those people who have made this dissertation possible. Someone I may have forgotten but I hope I have cited the most important ones. However, to all of them who have shared my social and scientific growth go my sincerely gratitude because my graduate experience has been one that I will cherish forever.