Technical University of Crete

Department of Electronic & Computer Engineering

Master of Science Thesis

"Novel and highly efficient reconfigurable implementations of Data Mining Algorithms"

Dagritzikos I. Panagiotis

Supervising Committee:

Ioannis Papaefstathiou, Associate Professor (Supervisor)

Apostolos Dollas, Professor

Minos Garofalakis, Professor

Chania, January 2012

To my family.

Thank you for all your love and support.

Abstract

Data mining is a relatively new field of computer science with a wide range of applications. The goal of data mining is to extract knowledge from huge data sets, like databases, in a human-understandable structure, like Decision Trees. This Master of Science thesis presents an innovative high-performance system level architecture for a state-of-the-art data mining algorithm on a modern FPGA. This is one of the first approaches utilizing the resources of an FPGA for accelerating certain very CPU intensive data-mining/data-classification schemes and as the real world results from actual runs on hardware demonstrate that it is a highly promising problem for reconfigurable technology.

The implemented system achieves performance speedup up to almost three orders of magnitude vs. the execution time of a well known Java platform of data mining (Weka), which is exectuted on a state-of-the-art multi-core CPU. Lastly, this work proposes a generic reconfigurable architecture that will be capable of constructing any kind of decision tree model independent to the input dataset's nature.

Acknowledgments

This senior thesis was elaborated at the Microprocessor and Hardware Laboratory (MHL) in partial fulfillment of the requirements for the degree of Master of Science from the department of Electronic and Computer Engineering of Technical University of Crete, under the supervision of the associate professor Ioannis Papaefstathiou.

First of all I would like to thank my advisor for the excellent working environment and the trust he granted for my research. I hope that we will have more chances to have a beer, without thinking of any demos!

Furthermore, I am grateful to Professor Apostolos Dollas and Professor Minos Garofalakis for their contribution and for their agreement to evaluate this thesis.

I would also like to mention all my colleagues in Microprocessor and Hardware Laboratory (MHL) who have greatly helped me to accomplish this work.

I am eternally grateful to my friend Gregory Chrysos for his endless support in the fulfillment of this thesis. He is a rare person, a great engineer and a great referee too!

I would also like to thank my friend Matina Lakka for her continuous support during these two years and for the wonderful time we had in Chania and while traveling abroad. I just wish I knew her during all those years I lived in Chania.

Moreover, I would like to thank my friends Konstantinos and Dimitris Makris for the great time we had all these years in Chania and for the many hours we spent fighting for no reason!

I would like to thank all my friends for all the unforgettable student years we had in Chania.

A big thank is not enough to express my gratitude to my parents Giannis and Maria and my sister Helen, who encouraged me to this attempt and mainly who supported and continue to support, by all means, me and my choices. This work is dedicated to them.

Contents

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Data mining has attracted a great deal of attention in the information industry and in scientific society in recent years, due to the wide availability of huge amounts of data and the imminent need for turning such data into useful information and knowledge. The information and knowledge, which is gained, can be used for applications ranging from market analysis, fraud detection, and customer retention, to production control and science exploration.

Data mining can be viewed as a result of the natural evolution of information technology. The database system industry has witnessed an evolutionary path in the development of the following functionalities: data collection and database creation, data management (including data storage and retrieval, and database transaction processing), and advanced data analysis (involving data warehousing and data mining). For instance, the early development of data collection and database creation mechanisms served as a prerequisite for later development of effective mechanisms for data storage and retrieval, and query and transaction processing. With numerous database systems offering query and transaction processing as common practice, advanced data analysis has naturally become the next target.

Simply stated, data mining refers to extracting or “mining” knowledge from large amounts of data. The term is actually a misnomer. Remember that the mining of gold from rocks or sand is referred to as gold mining rather than rock or sand mining. Thus, data mining should have been more appropriately named “knowledge mining from data,” which is unfortunately somewhat long. “Knowledge mining,” a shorter term, may not reflect the emphasis on mining from large amounts of data. Nevertheless, mining is a vivid term characterizing the process that finds a small set of precious nuggets from a great deal of raw material. Thus, such a misnomer that carries both “data” and “mining” became a popular choice. Many other terms carry a similar or slightly different meaning to data mining, such as knowledge mining from data, knowledge extraction, data/pattern analysis, data archaeology, and data dredging.

Many people treat data mining as a synonym for another popularly used term**, Knowledge Discovery from Data**, or KDD. Alternatively, others view data mining as simply an essential step in the process of knowledge discovery. Knowledge discovery as a process is depicted in Figure 1.1 and consists of an iterative sequence of the following steps:

1. Data cleaning (to remove noise and inconsistent data).
2. Data integration (where multiple data sources may be combined).
3. Data selection (where data relevant to the analysis task are retrieved from the database).
4. Data transformation (where data are transformed or consolidated into forms appropriate for mining by performing summary or aggregation operations, for instance).
5. Data mining (an essential process where intelligent methods are applied in order to extract data patterns).
6. Pattern evaluation (to identify the truly interesting patterns representing knowledge based on some interestingness measures).
7. Knowledge presentation (where visualization and knowledge representation techniques are used to present the mined knowledge to the user).

Steps 1 to 4 are different forms of data preprocessing, where the data are prepared for mining. The data mining step may interact with the user or a knowledge base. The interesting patterns are presented to the user and may be stored as new knowledge in the knowledge base. Note that according to this depiction, data mining is only one step in the entire process, but an essential one as it uncovers hidden patterns for evaluation.

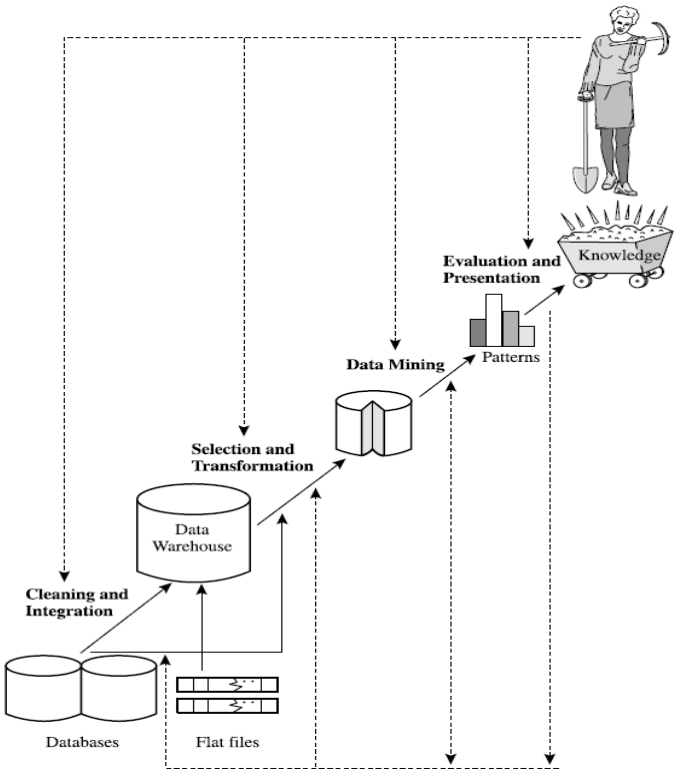

Figure 1-1: The knowledge discovery process

Data mining is a step in the knowledge discovery process. However, in industry, in media, and in the database research field, the term data mining is becoming more popular than the longer term of knowledge discovery from data. Therefore, the term data mining is chosen and a broad view of data mining functionality is adopted: data mining is the process of discovering interesting knowledge from large amounts of data stored in databases, data warehouses, or other information repositories.

Based on this view, the architecture of a typical data mining system may have the following major components:

- **Database, data warehouse, WorldWideWeb**, or other **information repository**: This is one or a set of databases, data warehouses, spreadsheets, or other kinds of information repositories. Data cleaning and data integration techniques may be performed on the data. Database or data warehouse server: The database or data warehouse server is responsible for fetching the relevant data, based on the user's data mining request.
- **Knowledge base:** This is the domain knowledge that is used to guide the search or evaluate the interesting resulting patterns. Such knowledge can include concept hierarchies,

used to organize attributes or attribute values into different levels of abstraction. Knowledge such as user beliefs, which can be used to assess an interesting pattern based on its unexpectedness, may also be included. Other examples of domain knowledge are additional interesting constraints or thresholds, and metadata (e.g., describing data from multiple heterogeneous sources).

- **Data mining engine:** This is essential to the data mining system and ideally consists of a set of functional modules for tasks such as characterization, association and correlation analysis, classification, prediction, cluster analysis, outlier analysis, and evolution analysis.
- **Pattern evaluation module:** This component typically employs interesting measures and interacts with the data mining modules so as to *focus* the search toward important patterns. It may use interesting thresholds to filter out discovered patterns. Alternatively, the pattern evaluation module may be integrated with the mining module, depending on the implementation of the data mining method used. For efficient data mining, it is highly recommended to push the evaluation of pattern "value" as deep as possible into the mining process so as to confine the search to only the interesting patterns.
- **User interface:** This module communicates between users and the data mining system, allowing the user to interact with the system by specifying a data mining query or task, providing information to help focus the search, and performing exploratory data mining based on the intermediate data mining results. In addition, this component allows the user to browse database and data warehouse schemas or data structures, evaluate mined patterns, and visualize the patterns in different forms.

From a data warehouse perspective, data mining can be viewed as an advanced stage of on-line analytical processing (OLAP). However, data mining goes far beyond the narrow scope of summarization-style analytical processing of data warehouse systems by incorporating more advanced techniques for data analysis.

Although there are many "data mining systems" on the market, not all of them can perform true data mining. A data analysis system that does not handle large amounts of data should be more appropriately categorized as a machine learning system, a statistical data analysis tool, or an experimental system prototype. A system that can only perform data or information retrieval, including finding aggregate values, or that performs deductive query answering in large databases should be more appropriately categorized as a database system, an information retrieval system, or a deductive database system.

Data mining involves an integration of techniques from multiple disciplines such as database and data warehouse technology, statistics, machine learning, high-performance computing, pattern recognition, neural networks, data visualization, information retrieval, image and signal processing, and spatial or temporal data analysis. As a result, emphasis is placed on *efficient* and *scalable* data mining techniques. For an algorithm to be scalable, its running time should grow approximately linearly in proportion to the size of the data, given the available system resources such as main memory and disk space.

By performing data mining, interesting knowledge, regularities, or high-level information can be extracted from databases and viewed or browsed from different angles. The discovered knowledge can be applied to decision making, process control, information management, and query processing. Therefore, data mining is considered one of the most important frontiers in database and information systems and one of the most promising interdisciplinary developments in the information technology.

Frequent patterns, as the name suggests, are patterns that occur frequently in data. There are many kinds of frequent patterns, including itemsets, subsequences, and substructures.

A **frequent itemset** typically refers to a set of items that frequently appear together in a transactional data set. A frequently occurring subsequence, such as the pattern that customers tend to purchase first a PC, followed by a digital camera, and then a memory card, is a (frequent) sequential pattern. A substructure can refer to different structural forms, such as graphs, trees, or lattices, which may be combined with itemsets or subsequences. If a substructure occurs frequently, it is called a (frequent) structured pattern. Mining frequent patterns leads to the discovery of interesting associations and correlations within data. Thus, the data mining scientific field can be divided into various categories with different characteristics.

The most important categories that will be examined in more details in chapter 2 are the following:

- **Association rule learning** is a popular method for discovering interesting relations between variables in large databases. Piatetsky-Shapiro [1] analyzes and presents strong rules that are discovered in databases using different measures of interest. Based on the concept of strong rules, Agrawal et al. [2] introduced association rules for discovering regularities between products in large scale transaction data recorded by point-of-sale (POS) systems in supermarkets.

- **Clustering** analyzes data objects without consulting a known class label. In general, the class labels are not present in the training data simply because they are not known. The objects are clustered or grouped based on the principle of maximizing the intraclass similarity and minimizing the interclass similarity. The clusters of objects are formed so that objects within a cluster have high similarity, but they must be very dissimilar to objects in other clusters. Each cluster that is formed can be viewed as a class of objects, from which rules can be derived. Clustering can also facilitate taxonomy formation, that is, the organization of observations into a hierarchy of classes that group similar events together.
- **Classification** is the process of finding a model (or function) that describes and distinguishes data classes or concepts. The goal of the classification process is to build a model that will predict the class of objects whose class label is unknown. The derived model is based on the analysis of a set of training data (i.e., data objects whose class label is known).
- **Regression analysis** is a statistical methodology that is most often used for numeric prediction, although other methods exist, as well. Prediction also encompasses the identification of distribution trends based on the available data.

The **contribution** of this work is in presenting an innovative reconfigurable logic based system that allows the construction of a Decision Tree; a data structure that belongs to the classification method as mentioned above. The main goal of this thesis is to accelerate the execution time of the decision tree construction method (DTC), as it is a time demanding procedure that requires even a few days for large input datasets to produce the final model. Two different implementations are presented that offer faster execution of the BFTree algorithm – a decision tree construction algorithm proposed in WEKA platform. The speedup of the implemented systems can reach up to approximately three times of magnitude faster than the execution of the corresponding software on a high end server. Moreover, based on the BFTree implementation, a more generic reconfigurable architecture of building a data mining decision tree is proposed, as there are only a few characteristics that change among the various existing DTC methods.

The rest of this thesis is organized as follows:

Chapter 2 explains the above mentioned terminology in details and mainly focuses on the decision tree construction methods and characteristics which are the basis of the reconfigurable system's implementation.

Chapter 3 briefly describes previous implementations on data mining algorithms and analyzes their basic characteristics and results.

Chapter 4 describes the data mining platform that was the case study in this thesis and explains the basic terminology and the algorithmic steps of the implemented classification decision tree.

Chapter 5 describes the two different architectures of the hardware implementation of the BFTree algorithm and makes a straight comparison between them.

Chapter 6 presents the performance results of the architectures, as well as a detailed comparison of the software runtime with the corresponding reconfiguration implementation

Chapter 7 concludes this thesis and proposes some possible extensions for future work.

2. Data Mining and Decision Trees

Data mining, as described in the previous chapter, consists of six common classes of tasks**: anomaly detection, association rule learning, clustering, classification, regression** and **summarization**. In a few words, anomaly detection is the identification of unusual data records or data errors that might be interesting and require further investigation. Association rule learning is the procedure of searching for relationships and dependencies between variables. Clustering is the task of discovering groups and structures in the data that are in some way or another "similar", without using known structures in the data. Classification is the task of generalizing known structure to apply to new data. Regression is the process of attempting to find a function which models the data with the least error. Finally, summarization is the provision of a more compact representation of the data set, including visualization and report generation. In this chapter, the three most important data mining tasks will be examined; **clustering, association rule learning** and **classification**.

2.1 CLUSTERING

The process of grouping a set of physical or abstract objects into classes of similar objects is called **clustering**. A cluster is a collection of data objects that are similar to one another within the same cluster and are dissimilar to the objects in other clusters. A cluster of data objects can be treated collectively as one group and so may be considered as a form of data compression. Although classification is an effective mean for distinguishing groups or classes of objects, it requires the often costly collection and labeling of a large set of training tuples or patterns, which the classifier uses to model each group. It is often more desirable to proceed in the reverse direction: first, partition the set of data into groups based on the data similarity (e.g., using clustering), and then assign labels to the relatively small number of groups. Additional advantages of such a clustering-based process are that it is adaptable to changes and it finds out useful features that distinguish different groups.

Cluster analysis is an important human activity. By automated clustering, dense and sparse regions in object space can be identified and, therefore, distribution patterns and interesting correlations among data attributes can be discovered. Cluster analysis has been widely used in numerous scientific areas, including market research, pattern recognition, data analysis, and image processing. In business, clustering can help marketers discover distinct groups in their customer bases and characterize customer groups based on purchasing patterns. In biology, it

can be used to derive plant and animal taxonomies, categorize genes with similar functionality, and gain insight into structures inherent in populations. The clustering may also help in the identification of areas of similar land use in an earth observation database and in the identification of groups of houses in a city according to house type, value, and geographic location, as well as the identification of groups of automobile insurance policy holders with a high average claim cost. It can also be used to help classify documents on the Web for information discovery.

The clustering is also called data segmentation in some applications because clustering partitions large data sets into groups according to their similarity. Clustering can also be used for outlier detection, where outliers (values that are “far away” from any cluster) may be more interesting than common cases. Applications of outlier detection include the detection of credit card fraud and the monitoring of criminal activities in electronic commerce.

As a data mining function, the cluster analysis can be used as a stand-alone tool to gain insight into the distribution of data, to observe the characteristics of each cluster, and to focus on a particular set of clusters for further analysis. Alternatively, it may serve as a preprocessing step for other algorithms, such as characterization, attribute subset selection, and classification, which would then operate on the detected clusters and the selected attributes or features.

Data clustering is under vigorous development and it contributes on many research areas, like data mining, statistics, machine learning, spatial database technology, biology, and marketing. The cluster analysis has recently become a highly active topic in data mining research due to the huge amounts of data collected in databases.

As a branch of statistics, the cluster analysis has been extensively studied for many years, focusing mainly on distance-based cluster analysis. Cluster analysis tools based on k-means, k-medoids, and several other methods have been built into many statistical analysis software packages or systems, such as S-Plus [3], SPSS [4], and SAS [5]. In machine learning, the clustering is an example of unsupervised learning. Unlike classification, clustering and unsupervised learning do not rely on predefined classes and class-labeled training examples. For this reason, clustering is a form of learning by observation, rather than learning by examples. In data mining, efforts have focused on finding methods for efficient cluster analysis in large databases. Active themes of research focus on the scalability of clustering methods, the effectiveness of methods for clustering complex shapes and types of data, high-dimensional

clustering techniques, and methods for clustering mixed numerical and categorical data in large databases.

Clustering is a challenging field of research in which its potential applications pose their own special requirements. The following are the typical requirements of clustering in data mining:

Scalability: Many clustering algorithms work well on small data sets containing fewer than several hundred data objects; however, a large database may contain millions of objects. Clustering on a sample of a given large data set may lead to biased results. As result, highly scalable clustering algorithms are needed.

Ability to deal with different types of attributes: Many algorithms are designed to cluster interval-based (numerical) data. However, applications may require clustering other types of data, such as binary, categorical (nominal), and ordinal data, or mixtures of these data types.

Discovery of clusters with arbitrary shape: Many clustering algorithms determine clusters based on Euclidean or Manhattan distance measurements. The algorithms that are based on such distance measures tend to find spherical clusters with similar size and density. However, a cluster could be of any shape. It is important to develop algorithms that can detect clusters of arbitrary shape.

Minimal requirements for domain knowledge to determine input parameters: Many clustering algorithms require users to input certain parameters in cluster analysis (such as the number of desired clusters). The clustering results can be quite sensitive to input parameters. Parameters are often difficult to determine, especially for data sets containing high-dimensional objects. This not only burdens users, but it also makes the quality of clustering difficult to control.

Ability to deal with noisy data: Most real-world databases contain outliers or missing, unknown, or erroneous data. Some clustering algorithms are sensitive to such data and may lead to clusters of poor quality.

Incremental clustering and insensitivity to the order of input records: Some clustering algorithms cannot incorporate newly inserted data (i.e., database updates) into existing clustering structures but they determine a new clustering from scratch. Some clustering algorithms are sensitive to the order of input data. That is, given a set of data objects, such an algorithm may return dramatically different clusterings depending on the order of presentation

of the input objects. It is important to develop incremental clustering algorithms and algorithms that are insensitive to the order of input.

High dimensionality: A database or a data warehouse can contain several dimensions or attributes. Many clustering algorithms are good at handling low-dimensional data, involving only two to three dimensions. Human eyes are good at judging the quality of clustering for up to three dimensions. Finding clusters of data objects in high dimensional space is challenging, especially considering that such data can be sparse and highly skewed.

Constraint-based clustering: Real-world applications may need to perform clustering under various kinds of constraints. Suppose that your job is to choose the locations for a given number of new automatic banking machines (ATMs) in a city. To decide upon this, you may cluster households while considering constraints such as the city's rivers and highway networks, and the type and number of customers per cluster. A challenging task is to find groups of data with good clustering behavior that satisfy specified constraints.

Interpretability and usability: Users expect clustering results to be interpretable, comprehensible, and usable. This is due to the fact that the clustering may need to be tied to specific semantic interpretations and applications. It is important to study how an application goal may influence the selection of clustering features and methods.

2.1.1 A Categorization of Major Clustering Methods

Many clustering algorithms exist in the literature. It is difficult to provide a crisp categorization of clustering methods because these categories may overlap, so that a method may have features from several categories. Nevertheless, it is useful to present a relatively organized picture of the different clustering methods. In general, the major clustering methods can be classified into the following categories.

Partitioning methods: Given a database of n objects or data tuples, a partitioning method constructs k partitions of the data, where each partition represents a cluster and $k \leq n$. Thus, it classifies the data into k groups, which together satisfy the following requirements:

- Each group must contain at least one object
- Each object must belong to exactly one group

Given k, the number of partitions to construct, a partitioning method creates an initial partitioning. It then uses an iterative relocation technique that attempts to improve the

partitioning by moving objects from one group to another. The general criterion of a good partitioning is that objects in the same cluster are “close” or related to each other, whereas objects of different clusters are “far apart” or very different. There are various kinds of other criteria for judging the quality of partitions. To achieve global optimality in partitioning-based clustering would require the exhaustive enumeration of all of the possible partitions. Instead, most applications adopt one of a few popular heuristic methods, such as (1) **the k-means algorithm** [6], where each cluster is represented by the mean value of the objects in the cluster, and (2) the **k-medoids algorithm** [7], where each cluster is represented by one of the objects located near the center of the cluster. These heuristic clustering methods work well for finding spherical-shaped clusters in small to medium-sized databases. The partitioning-based methods need to be extended in order to find clusters with complex shapes and for clustering very large data sets.

Hierarchical methods: A hierarchical method creates a hierarchical decomposition of the given set of data objects. A hierarchical method can be classified as being either agglomerative or divisive, based on how the hierarchical decomposition is formed. The agglomerative approach, also called the bottom-up approach, starts with each object forming a separate group. It successively merges the objects or groups that are close to one another, until all of the groups are merged into one (the topmost level of the hierarchy), or until a termination condition holds. The divisive approach, also called the top-down approach, starts with all of the objects in the same cluster. A cluster is split up into smaller clusters for each successive iteration, until eventually each object is in one cluster, or until a termination condition holds.

Hierarchical methods suffer from the fact that once a step (merge or split) is done, it can never be undone. This rigidity is useful in that it leads to smaller computation costs by not having to worry about a combinatorial number of different choices. However, such techniques cannot correct erroneous decisions.

There are two approaches to improving the quality of hierarchical clustering: (1) perform careful analysis of object “linkages” at each hierarchical partitioning, such as in *Chameleon* [8], or (2) integrate hierarchical agglomeration and other approaches by first using a hierarchical agglomerative algorithm to group objects into microclusters, and then performing macroclustering on the microclusters using another clustering method such as iterative relocation, as in *BIRCH* [9].

Density-based methods: Most partitioning methods cluster objects based on the distance between objects. Such methods can find only spherical-shaped clusters and encounter difficulty at discovering clusters of arbitrary shapes. Other clustering methods have been developed based on the notion of density. Their general idea is to continue growing the given cluster as long as the density (number of objects or data points) in the "neighborhood" exceeds some threshold; that is, for each data point within a given cluster, the neighborhood of a given radius has to contain at least a minimum number of points. Such a method can be used to filter out noise (outliers) and discover clusters of arbitrary shape. *DBSCAN* [10] and its extension, *OPTICS* [11], are typical density-based methods that grow clusters according to a density-based connectivity analysis. *DENCLUE* [12] is a method that clusters objects based on the analysis of the value distributions of density functions.

Grid-based methods: Grid-based methods quantize the object space into a finite number of cells that form a grid structure. All of the clustering operations are performed on the grid structure (i.e., on the quantized space). The main advantage of this approach is its fast processing time, which is typically independent of the number of data objects and dependent only on the number of cells in each dimension in the quantized space. *STING* [13] is a typical example of a grid-based method. *WaveCluster* [14] applies wavelet transformation for clustering analysis and is both grid-based and density-based.

Model-based methods: Model-based methods hypothesize a model for each of the clusters and find the best fit of the data to the given model. A model based algorithm may locate clusters by constructing a density function that reflects the spatial distribution of the data points. It also leads to a way of automatically determining the number of clusters based on standard statistics, taking "noise" or outliers into account and thus yielding robust clustering methods. *EM* [15] is an algorithm that performs expectation-maximization analysis based on statistical modeling. *COBWEB* [16] is a conceptual learning algorithm that performs probability analysis and takes concepts as a model for clusters. *SOM* [17] (or self-organizing feature map) is a neural network-based algorithm that clusters by mapping high dimensional data into a 2-D or 3-D feature map, which is also useful for data visualization.

The choice of clustering algorithm depends both on the type of data available and on the particular purpose of the application. If cluster analysis is used as a descriptive or exploratory tool, it is possible to try several algorithms on the same data to see what the data may disclose.

Some clustering algorithms integrate the ideas of several clustering methods, so that it is sometimes difficult to classify a given algorithm as uniquely belonging to only one clustering method category. Furthermore, some applications may have clustering criteria that require the integration of several clustering techniques. Aside from the above categories of clustering methods, there are two classes of clustering tasks that require special attention. One is clustering high-dimensional data, and the other is constraint-based clustering.

Clustering high-dimensional data is a particularly important task in cluster analysis because many applications require the analysis of objects containing a large number of features or dimensions. For example, text documents may contain thousands of terms or keywords as features, and DNA microarray data may provide information on the expression levels of thousands of genes under hundreds of conditions. Clustering high-dimensional data is challenging due to the curse of dimensionality.

Many dimensions may not be relevant. As the number of dimensions increases, the data become increasingly sparse so that the distance measurement between pairs of points become meaningless and the average density of points anywhere in the data is likely to be low. Therefore, a different clustering methodology needs to be developed for high-dimensional data.

CLIQUE [18] and *PROCLUS* [19] are two influential subspace clustering methods, which search for clusters in subspaces (or subsets of dimensions) of the data, rather than over the entire data space. Frequent pattern–based clustering, another clustering methodology, extracts distinct frequent patterns among subsets of dimensions that occur frequently. It uses such patterns to group objects and generate meaningful clusters. The *pCluster* [20] is an example of frequent pattern–based clustering that groups objects based on their pattern similarity.

Constraint-based clustering is a clustering approach that performs clustering by incorporation of user-specified or application-oriented constraints. A constraint expresses a user's expectation or describes "properties" of the desired clustering results, and provides an effective means for communicating with the clustering process. Various kinds of constraints can be specified, either by a user or as per application requirements.

In addition, *semi-supervised clustering* employs, for example, pairwise constraints (such as pairs of instances labeled as belonging to the same or different clusters) in order to improve the quality of the resulting clustering.

2.2 ASSOCIATION RULE MINING

Frequent patterns are patterns (such as itemsets, subsequences, or substructures) that appear in a data set frequently. For example, a set of items, such as milk and bread that appear frequently together in a transaction data set is a frequent itemset. A subsequence, such as buying first a PC, then a digital camera, and then a memory card, if it occurs frequently in a shopping history database, is a (frequent) sequential pattern. A substructure can refer to different structural forms, such as subgraphs, subtrees, or sublattices, which may be combined with itemsets or subsequences. If a substructure occurs frequently, it is called a (frequent) structured pattern. Finding such frequent patterns plays an essential role in mining associations, correlations, and many other interesting relationships among data. Moreover, it helps in data classification, clustering, and other data mining tasks as well. Thus, frequent pattern mining has become an important data mining task and a focused theme in data mining research.

Frequent itemset mining leads to the discovery of associations and correlations among items in large transactional or relational data sets. With massive amounts of data continuously being collected and stored, many industries are becoming interested in mining such patterns from their databases. The discovery of interesting correlation relationships among huge amounts of business transaction records can help in many business decision-making processes, such as catalog design, cross-marketing, and customer shopping behavior analysis. Among all algorithms that apply the association rule mining in data mining, Apriori [21] is the most important and popular one and it will be analyzed in the next section.

2.2.1 The Apriori Algorithm: Finding Frequent Itemsets Using Candidate Generation

Apriori is a seminal algorithm proposed by R. Agrawal and R. Srikant in 1994 for mining frequent itemsets for Boolean association rules. The name of the algorithm is based on the fact that the algorithm uses prior knowledge of frequent itemset properties.

Apriori employs an iterative approach known as a level-wise search, where k-itemsets are used to explore (k+1)-itemsets. First, the set of frequent 1-itemsets is found by scanning the database to accumulate the count for each item, and collecting those items that satisfy minimum support. The resulting set is denoted L1. Next, L1 is used to find L2, the set of frequent 2-itemsets, which is used to find L3, and so on, until no more frequent k-itemsets can be found. The finding of each L_k requires one full scan of the database. An important property, which is called the

Apriori property, presented below, is used to reduce the search space in order to improve the efficiency of the level-wise generation of frequent itemsets,.

Apriori property: All nonempty subsets of a frequent itemset must also be frequent. The Apriori property is based on the following observation. By definition, if an itemset I does not satisfy the minimum support threshold, min sup, then I is not frequent; that is, P(I) < min sup. If an item A is added to the itemset I, then the resulting itemset (i.e., I U A) cannot occur more frequently than I. Therefore, I U A is not frequent either; that is, P(I U A) < min sup.

This property belongs to a special category of properties called antimonotone in the sense that if a set cannot pass a test, all of its supersets will fail the same test as well. It is called antimonotone because the property is monotonic in the context of failing a test.

Many variations of the Apriori algorithm have been proposed that focus on improving the efficiency of the original algorithm. Several of these variations are summarized as follows:

Hash-based technique: A hash-based technique can be used to reduce the size of the candidate k-itemsets, Ck, for k > 1. For example, when scanning each transaction in the database to generate the frequent 1-itemsets, L1, from the candidate 1-itemsets in C1, all of the 2-itemsets can be generated for each transaction, hashed (i.e., mapped) into the different buckets of a hash table structure, and therefore, increase the corresponding bucket counts.

A 2-itemset whose corresponding bucket count in the hash table is below the support threshold cannot be frequent and thus should be removed from the candidate set. Such a hash-based technique may substantially reduce the number of the candidate k-itemsets examined (especially when k = 2).

Transaction reduction: A transaction that does not contain any frequent k-itemsets cannot contain any frequent (k+1)-itemsets. Therefore, such a transaction can be marked or removed from further consideration because subsequent scans of the database for j-itemsets, where j > k, will not require it.

Partitioning: A partitioning technique can be used that requires just two database scans to mine the frequent itemsets. It consists of two phases; In Phase I, the algorithm subdivides the transactions of D into n non overlapping partitions. If the minimum support threshold for transactions in D is min sup, then the minimum support count for a partition is min sup x the

number of transactions in that partition. For each partition, all frequent itemsets within the partition are found. These are referred to as local frequent itemsets.

The procedure employs a special data structure that, for each itemset, records the TIDs of the transactions containing the items in the itemset. This allows it to find all of the local frequent k-itemsets, for k = 1, 2, …, N , in just one scan of the database. A local frequent itemset may or may not be frequent with respect to the entire database, D. Any itemset that is potentially frequent with respect to D must occur as a frequent itemset in at least one of the partitions. Therefore, all local frequent itemsets are candidate itemsets with respect to D. The collection of frequent itemsets from all partitions forms the global candidate itemsets with respect to D.

In Phase II, a second scan of D is conducted in which the actual support of each candidate is assessed in order to determine the global frequent itemsets. Partition size and the number of partitions are set so that each partition can fit into main memory and therefore be read only once in each phase.

Sampling: The basic idea of the sampling approach is to pick a random sample S of the given data D, and then search for frequent itemsets in S instead of D. In this way, we trade off some degree of accuracy against efficiency. The sample size of S is such that the search for frequent itemsets in S can be done in main memory, and so only one scan of the transactions in S is required overall. Because we are searching for frequent itemsets in S rather than in D, it is possible that we will miss some of the global frequent itemsets. To lessen this possibility, a lower support threshold than minimum support is used to find the frequent itemsets local to S (denoted LS). The rest of the database is then used to compute the actual frequencies of each itemset in LS. A mechanism is used to determine whether all of the global frequent itemsets are included in LS. If LS actually contains all of the frequent itemsets in D, then only one scan of D is required. Otherwise, a second pass can be done in order to find the frequent itemsets that were missed in the first pass. The sampling approach is especially beneficial when efficiency is of utmost importance, such as in computationally intensive applications that must be run frequently.

2.2.2 Dynamic itemset counting (adding candidate itemsets at different points during a scan)

A dynamic itemset counting technique was proposed in which the database is partitioned into blocks marked by start points. In this variation, new candidate itemsets can be added at any start point, unlike in Apriori, which determines new candidate itemsets only immediately before each

complete database scan. The technique is dynamic as it estimates the support of all of the itemsets that have been counted so far, adding new candidate itemsets if all of their subsets are estimated to be frequent. The resulting algorithm requires fewer database scans than Apriori.

2.2.3 Mining Frequent Itemsets without Candidate Generation

In many cases the Apriori candidate generate-and-test method significantly reduces the size of candidate sets, leading to good performance gain. However, it can suffer from two nontrivial costs: It may need to generate a huge number of candidate sets. For example, if there are 10^4 frequent 1-itemsets, the Apriori algorithm will need to generate more than 10^7 candidate 2-itemsets. Moreover, to discover a frequent pattern of size 100, for example, it has to generate at least 10^{30} candidates in total. It may need to repeatedly scan the database and check a large set of candidates by pattern matching. It is costly to go over each transaction in the database to determine the support of the candidate itemsets.

An interesting method in this attempt is called frequent-pattern growth, or simply **FP-growth** [22], which adopts a divide-and-conquer strategy as follows. First, it compresses the database representing frequent items into a frequent-pattern tree, or FP-tree, which retains the itemset association information. It then divides the compressed database into a set of conditional databases (a special kind of projected database), each associated with one frequent item or "pattern fragment," and mines each such database separately.

2.2.4 Mining Frequent Itemsets Using Vertical Data Format

Both the Apriori and FP-growth methods mine frequent patterns from a set of transactions in TID-itemset format (that is, {TID : itemset}), where TID is a transaction-id and itemset is the set of items bought in transaction TID. This data format is known as horizontal data format. Alternatively, data can also be presented in item-TID set format (that is, {item : TID set}), where item is an item name, and TID set is the set of transaction identifiers containing the item. This format is known as vertical data format. The way that frequent itemsets can be mined efficiently using vertical data format is the essence of the **ECLAT** (Equivalence CLASS Transformation) algorithm developed by Zaki [23].

2.2.5 Mining Various Kinds of Association Rules

Efficient methods for mining frequent itemsets and association rules have been studied. In this section, additional application requirements are considered by extending the scope to include mining multilevel association rules, multidimensional association rules, and quantitative association rules in transactional and/or relational databases and data warehouses.

Multilevel association rules involve concepts at different levels of abstraction. Multidimensional association rules involve more than one dimension or predicate (e.g., rules relating what a customer buys as well as the customer's age.) Quantitative association rules involve numeric attributes that have an implicit ordering among values.

For many applications, it is difficult to find strong associations among data items at low or primitive levels of abstraction due to the sparsity of data at those levels. Strong associations discovered at high levels of abstraction may represent commonsense knowledge. Moreover, what may represent common sense to one user may seem novel to another. Therefore, the data mining systems should provide capabilities for mining association rules at multiple levels of abstraction, with sufficient flexibility for easy traversal among different abstraction spaces. A few categories for multi-level association rules are namely the following: *Mining multidimensional association rules from relational databases and data warehouses, mining multidimensional association rules using static discretization of quantitative attributes* and *finally mining quantitative association rules.*

2.3 CLASSIFICATION

Data classification is a process that has two basic steps. In the first step, a classifier is built describing a predetermined set of data classes or concepts. This is the learning step (or training phase), where a classification algorithm builds the classifier by analyzing or "learning from" a training set made up of database tuples and their associated class labels.

A tuple, $\boldsymbol{X}$, is represented by an n-dimensional attribute vector, $\boldsymbol{X} = (x_1, x_2, \ldots, x_n)$, depicting n measurements made on the tuple from n database attributes, respectively, $A_1, A_2, \ldots, An$. Each tuple, $\boldsymbol{X}$, is assumed to belong to a predefined class as determined by another database attribute called the class label attribute. The class label attribute is discrete-valued and unordered. It is *categorical* in that each value serves as a category or class. The individual tuples making up the training set are referred to as training tuples and are selected from the database under analysis. In the context of classification, data tuples can be referred to as *samples, examples, instances, data points,* or *objects*.

Because the class label of each training tuple *is provided*, this step is also known as supervised learning (i.e., the learning of the classifier is "supervised" in that it is told to which class each training tuple belongs). It contrasts with unsupervised learning (or clustering), in which the class label of each training tuple is not known, and the number or set of classes to be learned may not be known in advance.

This **first step** of the classification process can also be viewed as the learning of a mapping or function, $y = f(\boldsymbol{X})$, that can predict the associated class label y of a given tuple $\boldsymbol{X}$. In this view, we wish to learn a mapping or function that separates the data classes. Typically, this mapping is represented in the form of classification rules, decision trees, or mathematical formula. The rules can be used to categorize future data tuples, as well as provide deeper insight into the database contents. They also provide a compressed representation of the data.

In the **second step**, the model is used for classification. First, the predictive accuracy of the classifier is estimated. If we were to use the training set to measure the accuracy of the classifier, this estimate would likely be optimistic, because the classifier tends to overfit the data (i.e., during learning it may incorporate some particular anomalies of the training data that are not present in the general data set overall). Therefore, a test set is used, made up of test tuples and their associated class labels. These tuples are randomly selected from the general data set. They are independent of the training tuples, meaning that they are not used to construct the classifier.

The accuracy of a classifier on a given test set is the percentage of test set tuples that are correctly classified by the classifier. The associated class label of each test tuple is compared with the learned classifier's class prediction for that tuple. If the accuracy of the classifier is considered acceptable, the classifier can be used to classify future data tuples for which the class label is not known. Such data are also referred to in the machine learning literature as *"unknown"* or *"previously unseen"* data.

Data prediction is a two step process, similar to that of data classification. However, for prediction, we lose the terminology of "class label attribute" because the attribute for which values are being predicted is continuous-valued (ordered) rather than categorical (discrete-valued and unordered). The attribute can be referred to simply as the predicted attribute.

Prediction can also be viewed as a mapping or function, $y = f(\boldsymbol{X})$, where $\boldsymbol{X}$ is the input (e.g., a tuple describing a loan applicant), and the output y is a continuous or ordered value (e.g. the predicted amount that the bank can safely loan the applicant).

Prediction and classification also differ in the methods that are used to build their respective models. As with classification, the training set used to build a predictor should not be used to assess its accuracy. An independent test set should be used instead. The accuracy of a predictor

is estimated by computing an error based on the difference between the predicted value and the actual known value of *y* for each of the test tuples, ***X***.

2.3.1 Comparing Classification and Prediction Methods

Classification and prediction methods can be compared and evaluated according to the following criteria:

Accuracy: The accuracy of a classifier refers to the ability of a given classifier to correctly predict the class label of new or previously unseen data (i.e., tuples without class label information). Similarly, the accuracy of a predictor refers to how well a given predictor can guess the value of the predicted attribute for new or previously unseen data. As long as the accuracy computed is only an estimate of how well the classifier or predictor will do on new data tuples, confidence limits can be computed to help gauge this estimate.

Speed: This refers to the computational costs involved in generating and using the given classifier or predictor.

Robustness: This is the ability of the classifier or the predictor to make correct predictions given noisy data or data with missing values.

Scalability: This refers to the ability to construct the classifier or the predictor efficiently given large amounts of data.

Interpretability: This refers to the level of understanding and insight that is provided by the classifier or predictor. Interpretability is subjective and therefore more difficult to assess.

In the next sections, a few approaches on classification methods will be presented, concentrating mainly on the decision tree construction (DTC) methods that are the case study of this thesis.

2.3.2 Classification by Decision Tree Induction

Decision tree induction is the learning of decision trees from class-labeled training tuples. A decision tree is a flowchart-like tree structure, where each internal node (non leaf node) denotes a test on an attribute, each branch represents an outcome of the test, and each leaf node (or terminal node) holds a class label. The topmost node in a tree is the root node.

Given a tuple, **X**, for which the associated class label is unknown, the attribute values of the tuple are tested against the decision tree. A path is traced from the root to a leaf node, which

holds the class prediction for that tuple. Decision trees can easily be converted to classification rules.

The construction of decision tree classifiers does not require any domain knowledge or parameter setting, and therefore is appropriate for exploratory knowledge discovery. Decision trees can handle high dimensional data. Their representation of acquired knowledge in tree form is intuitive and generally easy to assimilate by humans. The learning and classification steps of decision tree induction are simple and fast. In general, decision tree classifiers have good accuracy. However, successful use may depend on the data at hand. Decision tree induction algorithms have been used for classification in many application areas, such as medicine, manufacturing and production, financial analysis, astronomy, and molecular biology. Decision trees are the basis of several commercial rule induction systems.

During tree construction, attribute selection measures are used to select the attribute that best partitions the tuples into distinct classes. When decision trees are built, many of the branches may reflect noise or outliers in the training data. Tree pruning attempts to identify and remove such branches, with the goal of improving classification accuracy on unseen data. Tree pruning is described more analytically in the next chapter.

2.3.2.1 Decision Tree Induction

During the late 1970s and early 1980s, J. Ross Quinlan, a researcher in machine learning, developed a decision tree algorithm known as ID3 (Iterative Dichotomiser) [24]. This work expanded on earlier work on *concept learning systems*, described by E. B. Hunt, J. Marin, and P. T. Stone [25]. Quinlan later presented C4.5 [26] (a successor of ID3), which became a benchmark to which newer supervised learning algorithms are often compared.

In 1984, a group of statisticians (L. Breiman, J. Friedman, R. Olshen, and C. Stone) published the book *Classification and Regression Trees* (CART) [27], which described the generation of binary decision trees. ID3 and CART were invented independently of one another at around the same time, yet follow a similar approach for learning decision trees from training tuples. These two cornerstone algorithms spawned a flurry of work on decision tree induction.

ID3, C4.5, and CART adopt a greedy (i.e., non backtracking) approach in which decision trees are constructed in a top-down recursive divide-and-conquer manner. Most algorithms for decision tree induction also follow such a top-down approach, which starts with a training set of

tuples and their associated class labels. The training set is recursively partitioned into smaller subsets as the tree is being built.

The strategy of the algorithm is as follows; the algorithm is called with three parameters: *D*, *attribute list*, and *Attribute selection method*. *D is referred* as a data partition. Initially, it is the complete set of training tuples and their associated class labels. The parameter *attribute list* is a list of attributes describing the tuples. *Attribute selection method* specifies a heuristic procedure for selecting the attribute that "best" discriminates the given tuples according to class. This procedure employs an attribute selection measure, such as information gain or the gini index. Whether the tree is strictly binary is generally driven by the attribute selection measure. Some attribute selection measures, such as the gini index, enforce the resulting tree to be binary. Others, like information gain, do not, therein allowing multiway splits (i.e., two or more branches to be grown from a node).

The tree starts as a single node, *N*, representing the training tuples in *D*. If the tuples in *D* are all of the same class, then node *N* becomes a leaf and is labeled with that class. Otherwise, the algorithm calls the corresponding *Attribute selection method* to determine the splitting criterion. The splitting criterion tells us which attribute to test at node *N* by determining the "best" way to separate or partition the tuples in *D* into individual classes.

The splitting criterion also tells us which branches to grow from node *N* with respect to the outcomes of the chosen test. More specifically, the splitting criterion indicates the splitting attribute and may also indicate either a split-point or a splitting subset. The splitting criterion is determined so that, ideally, the resulting partitions at each branch are as "pure" as possible. A partition is pure if all of the tuples in it belong to the same class. In other words, if we were to split up the tuples in *D* according to the mutually exclusive outcomes of the splitting criterion, we hope for the resulting partitions to be as pure as possible.

The node *N* is labeled with the splitting criterion, which serves as a test at the node. A branch is grown from node *N* for each of the outcomes of the splitting criterion. The tuples in *D* are partitioned accordingly.

There are three possible scenarios, as analyzed above. Let *A* be the splitting attribute. *A* has *v* distinct values, $\{a_1, a_2, \ldots, a_v\}$, based on the training data.

- *A is discrete-valued*: In this case, the outcomes of the test at node *N* correspond directly to the known values of *A*. A branch is created for each known value, a_j,

of A and labeled with that value. Partition D_j is the subset of class-labeled tuples in D having value a_j of A. Because all of the tuples in a given partition have the same value for A, then A need not be considered in any future partitioning of the tuples. Therefore, it is removed from *attribute list*.

- *A is continuous-valued*: In this case, the test at node N has two possible outcomes, corresponding to the conditions $A \leq$ *split point* and $A >$ *split point*, respectively, where *split point* is the split-point returned by *Attribute selection method* as part of the splitting criterion. (In practice, the split-point, a, is often taken as the midpoint of two known adjacent values of A and therefore may not actually be a pre-existing value of A from the training data.) Two branches are grown from N and labeled according to the above outcomes. The tuples are partitioned such that D_1 holds the subset of class-labeled tuples in D for which $A \leq$ *split point*, while D_2 holds the rest.
- *A is discrete-valued* and a *binary tree* must be produced (as dictated by the attribute selection measure or algorithm being used): The test at node N is of the form "$A \in S_A$". S_A is the splitting subset for A, returned by *Attribute selection method* as part of the splitting criterion. It is a subset of the known values of A. If a given tuple has value a_j of A and if $a_j \in S_A$, then the test at node N is satisfied. Two branches are grown from N. By convention, the left branch out of N is labeled *yes* so that $D1$ corresponds to the subset of class-labeled tuples in D that satisfy the test. The right branch out of N is labeled *no* so that $D2$ corresponds to the subset of class-labeled tuples from D that do not satisfy the test.

The algorithm uses the same process recursively to form a decision tree for the tuples at each resulting partition, D_j, of D. The recursive partitioning stops only when any one of the following terminating conditions is true:

- All of the tuples in partition D (represented at node N) belong to the same class
- There are no remaining attributes on which the tuples may be further partitioned. In this case, majority voting is employed. This involves converting node N into a leaf and labeling it with the most common class in D. Alternatively, the class distribution of the node tuples may be stored.
- There are no tuples for a given branch, that is, a partition D_j is empty. In this case, a leaf is created with the majority class in D.

Thus, the resulting decision tree is returned. The computational complexity of the algorithm given training set D is $O(n \times |D| \times log(|D|))$, where n is the number of attributes describing the tuples in D and $|D|$ is the number of training tuples in D. This means that the computational cost of growing a tree grows at most $n \times |D| \times log(|D|)$ with $|D|$ tuples.

Incremental versions of decision tree induction have also been proposed. When given new training data, these restructure the decision tree acquired from learning on previous training data, rather than relearning a new tree from scratch. Differences in decision tree algorithms include how the attributes are selected in creating the tree and the mechanisms used for pruning. The basic algorithm described above requires one pass over the training tuples in D for each level of the tree. This can lead to long training times and lack of available memory when dealing with large databases.

2.3.3 Bayesian Classification

Bayesian classifiers are statistical classifiers. They can predict class membership probabilities, such as the probability that a given tuple belongs to a particular class. Bayesian classification is based on Bayes' theorem. Studies comparing classification algorithms have found a simple Bayesian classifier known as the naïve Bayesian classifier to be comparable in performance with decision tree and selected neural network classifiers. Bayesian classifiers have also exhibited high accuracy and speed when applied to large databases.

Naïve Bayesian classifiers assume that the effect of an attribute value on a given class is independent of the values of the other attributes. This assumption is called class conditional independence. It is made to simplify the computations involved and, in this sense, is considered "naïve." Bayesian belief networks are graphical models, which unlike naïve Bayesian classifiers, allow the representation of dependencies among subsets of attributes. Bayesian belief networks can also be used for classification.

2.3.4 Rule-Based Classification

In rule-based classifiers the learned model is represented as a set of IF-THEN rules. There are several ways in which these rules can be generated, either from a decision tree or directly from the training data using a sequential covering algorithm.

2.3.5 Classification by back propagation

Backpropagation is a neural network learning algorithm. The field of neural networks was originally kindled by psychologists and neurobiologists who sought to develop and test

computational analogues of neurons. Roughly speaking, a neural network is a set of connected input/output units in which each connection has a weight associated with it. During the learning phase, the network learns by adjusting the weights so as to be able to predict the correct class label of the input tuples. Neural network learning is also referred to as connectionist learning due to the connections between units.

Neural networks involve long training times and are therefore more suitable for applications where this is feasible. They require a number of parameters that are typically best determined empirically, such as the network topology or "structure." Neural networks have been criticized for their poor interpretability. For example, it is difficult for humans to interpret the symbolic meaning behind the learned weights and of "hidden units" in the network. These features initially made neural networks less desirable for data mining.

Advantages of neural networks, however, include their high tolerance of noisy data as well as their ability to classify patterns on which they have not been trained. They can be used when you may have little knowledge of the relationships between attributes and classes. They are well-suited for continuous-valued inputs and outputs, unlike most decision tree algorithms. They have been successful on a wide array of real-world data, including handwritten character recognition, pathology and laboratory medicine, and training a computer to pronounce English text. Neural network algorithms are inherently parallel; parallelization techniques can be used to speed up the computation process.

In addition, several techniques have recently been developed for the extraction of rules from trained neural networks. These factors contribute toward the usefulness of neural networks for classification and prediction in data mining. There are many different kinds of neural networks and neural network algorithms. The most popular neural network algorithm is back propagation [28], which gained repute in the 1980s.

2.3.6 Support Vector Machines

Support Vector Machines is a promising new method for the classification of both linear and nonlinear data. In a nutshell, a support vector machine (or SVM) is an algorithm that works as follows. It uses a nonlinear mapping to transform the original training data into a higher dimension. Within this new dimension, it searches for the linear optimal separating hyperplane (that is, a "decision boundary" separating the tuples of one class from another). With an appropriate nonlinear mapping to a sufficiently high dimension, data from two classes can

always be separated by a hyperplane. The SVM finds this hyperplane using *support vectors* ("essential" training tuples) and *margins* (defined by the support vectors).

The first paper on support vector machines was presented in 1992 by Vladimir Vapnik and colleagues Bernhard Boser and Isabelle Guyon [29], although the groundwork for SVMs has been around since the 1960s (including early work by Vapnik and Alexei Chervonenkis on statistical learning theory [30]). Although the training time of even the fastest SVMs can be extremely slow, they are highly accurate, owing to their ability to model complex nonlinear decision boundaries. They are much less prone to overfitting than other methods. The support vectors found also provide a compact description of the learned model. SVMs can be used for prediction as well as classification. They have been applied to a number of areas, including handwritten digit recognition, object recognition, and speaker identification, as well as benchmark time-series prediction tests.

2.3.7 Associative Classification: Classification by Association Rule Analysis

Frequent patterns and their corresponding association or correlation rules characterize interesting relationships between attribute conditions and class labels, and thus have been recently used for effective classification. Association rules show strong associations between attribute-value pairs (or *items*) that occur frequently in a given data set. Association rules are commonly used to analyze the purchasing patterns of customers in a store. Such analysis is useful in many decision-making processes, such as product placement, catalog design, and cross-marketing. The discovery of association rules is based on *frequent itemset mining.*

The general idea is that we can search for strong associations between frequent patterns (conjunctions of attribute-value pairs) and class labels. Because association rules explore highly confident associations among multiple attributes, this approach may overcome some constraints introduced by decision-tree induction, which considers only one attribute at a time. In many studies, associative classification has been found to be more accurate than some traditional classification methods, such as C4.5 [26]. In particular, there are three main methods: CBA [31], CMAR [32], and CPAR [33].

2.3.8 Lazy Learners

The classification methods discussed so far in this chapter—decision tree induction, Bayesian classification, rule-based classification, classification by back propagation, support vector machines, and classification based on association rule mining—are all examples of *eager*

learners. Eager learners, when given a set of training tuples, will construct a generalization (i.e., classification) model before receiving new (e.g., test) tuples to classify.

Imagine a contrasting lazy approach, in which the learner instead waits until the last minute before doing any model construction in order to classify a given test tuple. That is, when given a training tuple, a lazy learner simply stores it (or does only a little minor processing) and waits until it is given a test tuple. Only when it sees the test tuple does it perform generalization in order to classify the tuple based on its similarity to the stored training tuples. Unlike eager learning methods, lazy learners do less work when a training tuple is presented and more work when making a classification or prediction. Because lazy learners store the training tuples or "instances," they are also referred to as instance based learners, even though all learning is essentially based on instances.

When making a classification or prediction, lazy learners can be computationally expensive. They require efficient storage techniques and are well-suited to implementation on parallel hardware. They offer little explanation or insight into the structure of the data. Lazy learners, however, naturally support incremental learning. They are able to model complex decision spaces having hyperpolygonal shapes that may not be as easily describable by other learning algorithms (such as hyper-rectangular shapes modeled by decision trees). In this section, we look at two examples of lazy learners: *k-nearest neighbor classifiers* and *case-based reasoning classifiers*.

K*-Nearest-Neighbor Classifiers:** Nearest-neighbor classifiers are based on learning by analogy, that is, by comparing a given test tuple with training tuples that are similar to it. The training tuples are described by *n* attributes. Each tuple represents a point in an *n*-dimensional space. In this way, all of the training tuples are stored in an *n*-dimensional pattern space. When given an unknown tuple, a ***k-nearest neighbor classifier searches the pattern space for the *k* training tuples that are closest to the unknown tuple. These *k* training tuples are the *k* "nearest neighbors" of the unknown tuple.

Case-Based Reasoning Classifiers: Case-based reasoning (CBR) classifiers use a database of problem solutions to solve new problems. Unlike nearest-neighbor classifiers, which store training tuples as points in Euclidean space, CBR stores the tuples or "cases" for problem solving as complex symbolic descriptions. Business applications of CBR include problem resolution for customer service help desks, where cases describe product-related diagnostic problems. CBR has also been applied to areas such as engineering and law, where cases are

either technical designs or legal rulings, respectively. Medical education is another area for CBR, where patient case histories and treatments are used to help diagnose and treat new patients.

2.3.9 Other Classification Methods

Several other classification methods exist, including genetic algorithms, rough set approach, and fuzzy set approaches. In general, these methods are less commonly used for classification in commercial data mining systems than the methods described earlier in this chapter.

2.4 PLATFORMS

There are several software suites for data mining, which are available in the web, and they cover all the above mentioned algorithmic categories. In this section, a brief summary of the most important data mining platforms that are either for commercial purpose or free will be made.The most famous platforms and their characteristics are the following;

AdvancedMiner (formerly Gornik), a platform for data mining and analysis, featuring modeling interface (OOP script, latest GUI design, advanced visualization) and grid computing.

BayesiaLab, a complete and powerful data mining tool based on Bayesian networks, including data preparation, missing values imputation, data and variables clustering, unsupervised and supervised learning.

Data Miner Software Kit, a collection of data mining tools, offered in combination with a book: Predictive Data Mining: A Practical Guide, Weiss and Indurkhya.

IBM Intelligent Miner Data Mining Suite, now fully integrated into the IBM InfoSphere Warehouse software; includes Data and Text mining tools (based on UIMA).

Nuggets, builds models that uncover hidden facts and relationships, predict for new data, and find key variables (Windows).

Pentaho, an open-source BI suite, including reporting, analysis, dashboards, data integration, and data mining based on Weka.

SAS Enterprise Miner, an integrated suite which provides a user-friendly GUI front-end to the SEMMA (Sample, Explore, Modify, Model, Assess) process.

Statistica Data Miner, a comprehensive, integrated statistical data analysis, graphics, data base management, and application development system.

Synapse, a development environment for neural networks and other adaptive systems, supporting the entire development cycle from data import and preprocessing via model construction and training to evaluation and deployment; allows deployment as .NET components.

All the above platforms are commercial and their prices range from a few hundred to thousand euros. Some free or shareware data mining platforms are the following:

AlphaMiner, an open source data mining platform that offers various data mining model building and data cleansing functionality.

ELKI (Environment for DeveLoping KDD-Applications Supported by Index-Structures), a framework in Java which includes clustering, outlier detection, and other algorithms; allows user to evaluate the combination of arbitrary algorithms, data types, and distance functions.

MiningMart, a graphical tool for data preprocessing and mining on relational databases; supports development, documentation, re-use and exchange of complete KDD processes. Free for non-commercial purposes.

RapidMiner, a leading open-source system for knowledge discovery and data mining.

TANAGRA, offers a GUI interface and methods for data access, statistics, feature selection, classification, clustering, visualization, association and more.

Weka, a collection of machine learning algorithms for solving real-world data mining problems. It is written in Java and runs on almost any platform.

2.5 DATASETS

There are several datasets in the web that are available for data miners, offering a wide variety of test case characteristics. Thus, datasets with variable number of attributes and instances, covering a wide range of scientific fields can be used in order to fully test and observe the data mining platforms' performance and correct operation. An indicative example of some data mining datasets is given below:

KDD Cup center, with all data, tasks, and results.

UCI KDD Database Repository for large datasets used in machine learning and knowledge discovery research.

AWS (Amazon Web Services) Public Data Sets, provides a centralized repository of public data sets that can be seamlessly integrated into AWS cloud-based applications.

Bioassay data, described in Virtual screening of bioassay data, by Amanda Schierz, J. of Cheminformatics, with 21 Bioassay datasets (Active / Inactive compounds) available for download.

Canada Open Data, pilot project with many government and geospatial datasets.

Data Source Handbook, A Guide to Public Data, by Pete Warden, O'Reilly (Jan 2011).

Data.gov.uk, publicly available data from UK (also London datastore.)

DataMarket, visualize the world's economy, societies, nature, and industries, with 100 million time series from UN, World Bank, Eurostat and other important data providers.

DataSF.org, a clearinghouse of datasets available from the City & County of San Francisco, CA.

DataFerrett, a data mining tool that accesses and manipulates TheDataWeb, a collection of many on-line US Goverment datasets.

EconData, thousands of economic time series, produced by a number of US Government agencies.

Enron Email Dataset, data from about 150 users, mostly senior management of Enron.

FEDSTATS, a comprehensive source of US statistics and more

FIMI repository for frequent itemset mining, implementations and datasets.

Financial Data Finder at OSU, a large catalog of financial data sets

GEO (GEO Gene Expression Omnibus), a gene expression/molecular abundance repository supporting MIAME compliant data submissions, and a curated, online resource for gene expression data browsing, query and retrieval.

GeoDa Center, geographical and spatial data.

Google ngrams datasets, text from millions of books scanned by Google.

Hilary Mason research-quality Big Data sets collection - many text and image datasets.

ICWSM-2009 dataset contains 44 million blog posts made between August 1st and October 1st, 2008.

Infobiotics PSP (protein structure prediction) datasets, adjustable real-world family of benchmarks for testing the scalability of classification/regression methods.

Infochimps, an open catalog and marketplace for data. You can share, sell, curate, and download data about anything and everything.

MIT Cancer Genomics gene expression datasets and publications, from MIT Whitehead Center for Genome Research.

ML Data, the data repository of the EU Pascal2 networks.

NASDAQ Data Store, provides access to market data.

National Government Statistical Web Sites, data, reports, statistical yearbooks, press releases, and more from about 70 web sites, including countries from Africa, Europe, Asia, and Latin America.

National Space Science Data Center (NSSDC), NASA data sets from planetary exploration, space and solar physics, life sciences, astrophysics, and more.

OpenData from Socrata, access to over 10,000 datasets including business, education, government, and fun.

PubGene(TM) Gene Database and Tools, genomic-related publications database

Robert Schiller data on housing, stock market, and more from his book Irrational Exuberance.

SMD: Stanford Microarray Database, stores raw and normalized data from microarray experiments.

SourceForge.net Research Data, includes historic and status statistics on approximately 100,000 projects and over 1 million registered users' activities at the project management web site.

Wikipedia User Contribution Dataset, prepared for an ongoing study on user reputation and content quality in Wikipedia at UCI.

Wikiposit, a (virtual) amalgamation of (mostly financial) data from many different sites, allowing users to merge data from different sources

Yahoo Sandbox datasets, Language, Graph, Ratings, Advertising and Marketing, Competition datasets.

3. RELATED WORK

Data mining, as described above, is a suitable field for hardware acceleration. Many hardware implementations on various data mining algorithms have been recently presented. This section describes some works that implement data mining algorithms on reconfigurable hardware platforms. As it shown in the following sections, there are several implementations that solve different problems of the data mining area and they are implemented on various hardware platforms, such as FPGAs and GPUs. The discrimination of these works is made according to the data mining field that they refer to as long as the platform where they are implemented.

3.1 Association algorithms

In this work made by **Sun and Zambreno** [34], the original scheme introduced in [35]- a reconfigurable systolic tree architecture for frequent pattern mining - is modified by eliminating the counting nodes, and provide a count mode algorithm.

Similar to the FP-growth algorithm, two scans of the transactional database are required. In the first scan both the set of frequent items and the support count of each frequent item is collected. This task is implemented in the software component of the system as shown in Fig. 4-1. Each candidate frequent itemset is dictated by the software module into the systolic tree in order to mine the frequent itemset. The software module and systolic tree communicates through a Processor Local Bus (PLB). The systolic tree then reports the support count of the itemset back to the software module. The process of producing candidate itemsets is critical to the runtime of the whole system. When selecting candidate itemsets to dictate, those itemsets which are not frequent are discarded.

The software controller is also responsible for transforming the transactional database onto the systolic tree. The FPGA-based hardware component of the embedded platform is responsible for building the systolic tree while receiving the transactions sent by the software module, and extracting the support count of the candidate itemsets dictated by the software module.

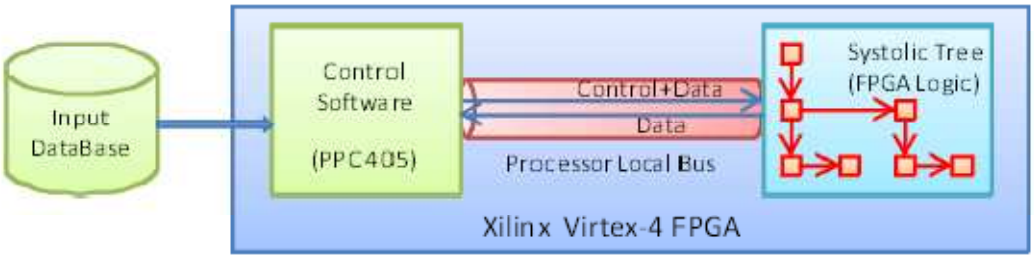

Figure 3-1: The overall system of [35]

This architecture implements small processing elements that map to an FPGA-based embedded system. Both the hardware space requirement and the frequent pattern mining time are entirely dependent on the size of the systolic tree. With carefully selected tree size, the mining time of systolic tree can be orders of magnitude faster than the FP-tree. Due to the structural characteristic of the proposed systolic tree architecture, the tree size cannot be very large.

In [36], **Baker and Prasanna** proposed a highly parallel custom architecture implemented on a reconfigurable computing system for the Apriori algorithm. Using a "bitmapped Content Addressable Memory (CAM)," the time and area required for executing the subset operations fundamental to data mining can be significantly reduced. The bitmapped CAM architecture implementation on an FPGA-accelerated high performance workstation provides a performance acceleration of orders of magnitude over software-based systems.

The bitmapped CAM utilizes redundancy within the candidate data to efficiently store and process many subset operations simultaneously. The efficiency of this operation allows 140 units to process about 2,240 subset operations simultaneously. Using industry-standard benchmarking databases, the implemented design offers a minimum of 24x time performance advantage over the fastest software Apriori implementations.

In [37], **Baker and Prasanna** also presented an alternative approach for implementing Apriori algorithm with systolic arrays. In this architecture, more than 500 units are mapped on a single FPGA device. Their results are based on the place-and-route of the full systolic array design on a Xilinx Virtex-II Pro 100 device with 44,000 slices. Hardware usage is 70 slices per systolic unit with resources for up to 16 2-byte item candidate sets. The units are all connected end-to-end in the form of a linear array. Each unit contains memory locations to temporarily store the candidates whose support is being calculated and to allow for stalling. A unit is composed of the

candidate memory, an index counter, and a comparator, which allows the output of the candidate memory to be compared with an incoming item.

The proposed architecture provides a performance improvement that can be orders of magnitude faster than the state-of-the-art software implementations. The system is scalable and introduces an efficient systolic injection method for intelligently reporting unpredictably generated mid-array results to a controller without any chance of collision or excessive stalling.

3.2 Clustering

K-means clustering is a very popular clustering technique, which is used in numerous applications, including data mining. In the k-means clustering algorithm, each point in the dataset is assigned to the nearest cluster by calculating distances from each point to the cluster centers. Then, the cluster centers are improved so that the error (sum of the square distances) is minimized. These two steps, referred as iteration, are repeated until no improvement of the error is obtained. The computation of the distances is a very time-consuming task, particularly for large dataset and large number of clusters. With dedicated hardware systems, the performance of the distance calculations can be accelerated by processing them in parallel. However, when the number of clusters is large, its performance is not enough for real-time applications, because the distances to all cluster centers cannot be calculated in parallel with reasonable amount of hardware resources. In order to reduce the computation time, many sophisticated software algorithms have been proposed to date. However, it is not easy to implement those techniques on the hardware systems because of their complexity.

In [38], **Saegusa and Maruyama** proposed a hardware implementation of k-means algorithm. They showed that real-time k-means clustering can be realized for large-size color images and large number of clusters, by generating kd-trees dynamically on FPGA, and by reducing the amount of computation of squared distances using the kd-trees. By reusing units to calculate squared distances for generating kd-trees and for assigning pixels to one of the K clusters, the whole circuit can be implemented on one FPGA. With the current largest FPGA with DDR memory interface, it will be possible to improve the performance by processing more pixels in parallel. The performance of the system is more than 30 fps in average (all images cannot be processed within 1/30 s), a performance fast enough for video sequences, when the number of pixels in an image is not larger than 300 Kb.

In [39], another implementation of k-means clustering algorithm was proposed by **Leeser** and **Szymanski**. In mapping the k-means algorithm to FPGA hardware, they examined algorithm level transforms that dramatically increased the achievable parallelism. They applied the k-means algorithm to multi-spectral and hyper-spectral images, which have tens to hundreds of channels per pixel of data.

K-means is a common solution to the segmentation of multi-dimensional data. The standard software implementation of k-means uses floating-point arithmetic and Euclidean distances. Floating point arithmetic and the multiplication- heavy Euclidean distance calculation are fine on a general purpose processor, but they have large area and speed penalties when implemented on an FPGA. In order to get the best performance of k-means on an FPGA, the algorithm needs to be transformed to eliminate these operations. Thus, they examined the effects of using two other distance measures, Manhattan and Max that do not require multipliers. They also examined the effects of using fixed precision and truncated bit widths in the algorithm. As a result of the analysis made in the algorithm, their implementation exhibits approximately a 200 times speed up over a software implementation and when downloaded to an Annapolis Wildstar board, a speed up of two orders of magnitude over the software implementation was achieved.

3.3 Classification

One important problem in data mining is Classification, which is the task of assigning objects to one of several predefined categories. Among the several solutions developed, Decision Tree Classification (DTC) is a popular method that yields high accuracy while handling large datasets. However, DTC is a computationally intensive algorithm, and as data sizes increase, its running time can stretch to several hours.

In [40], **Narayanan and Zambreno** proposed a hardware implementation of Decision Tree Classification. They identified the compute intensive kernel (Gini Score computation) in the algorithm, and developed an architecture, which was further optimized by reordering and simplifying the computations and by using a bitmapped data structure. The Gini score is a mathematical measure of the inequality of a distribution and calculating the Gini value for a particular split index involves computing the frequency of each class in each of the partitions.

The DTC architecture was implemented on a Xilinx Virtex-II Pro-based embedded development platform. The DTC unit was implemented as a custom peripheral which was fed

by the PowerPC. The PowerPC read in input data stored in DDR DIMM, initialized the DTC component, and supplied class ID data at regular intervals. The OCM BRAM block stores the instructions for the PowerPC operation.

While implementing the design, several tradeoffs were considered. The use of floating point computations complicated the design and increased the area overhead, hence they decided to perform the division operations using only fixed-point integer computations. To verify the correctness of the assumptions made, they implemented a version of ScalParC that uses only fixed point values. It was found that the decision trees generated by both the fixed-point and floating point versions were identical, thus validating their choice of a divider performing fixed point computations. The divider output was configured to produce 32 integer bits and 16 fractional bits, a choice made keeping in mind the size of the dataset and precision required to produce accurate results. The divider was also pipelined in order to handle multiple input class IDs at the same time.

The IBM Data Quest Generator was used to generate the data used in their performance measurements. The Gini calculation was also implemented in software (using C) and ran on the PowerPC under identical conditions. The speedup provided by hardware was measured in terms of the ratio of number of cycles taken by the hardware-enabled design to those taken by the software implementation. Figure 3 shows the speedups obtained when the DTC module was tested on the FPGA. The results show significant speedups over software implementations. As expected, the speedup increases with the number of Gini units on board, due to the parallelism offered by additional hardware computation units.

The experimental hardware imposed a size limitation of 16 Gini units, which achieves a speedup of 5.58×. It would be possible to achieve larger speedups using higher-capacity FPGAs. Given the fraction of execution time that the Gini score calculation takes in **ScalParC** [41, 42], the overall speedup of this particular implementation of DTC can be estimated to be 1.5×.

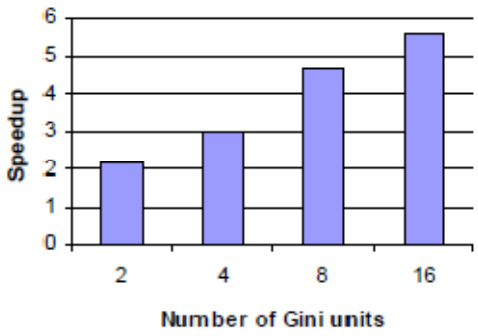

Figure 3-2: [40] system performance

In [43], **Struharik** and **Novak** designed eight Intellectual Property (IP) cores, two cores (non-programmable and programmable) for each of the four proposed architectures in [44] and [45]. To be able to serve as a general purpose building block for System on Chip (SoC) designs, IP core implementation of a Decision Tree should meet a range of application requirements. This fact excludes the usage of hard cores. Therefore proposed DT IP cores have been realized as soft IP cores using a synthesizable RTL description of the design. Every core comes with a set of parameters offering the opportunity for a designer to address a range of application problems with differing structural requirements (number of problem attributes, structure of the DT, number of DT nodes, depth of the DT, number formats for representation of attribute values, input size and characteristics at each node, class membership values etc.) and performance requirements (clock period, power dissipation and area).

They used the well-known XOR classification problem [46] to illustrate how to configure DT cores. This problem is described by two numerical attributes and four 2-bit XOR instances, which must be classified into one of two classes. Experimental results obtained on 23 data sets of standard UCI machine learning repository database suggest that the proposed architecture based on the sequence of UNs requires on average 56% less hardware resources compared with architectures [44, 45], while having the same throughput.

3.4 GPUs

Modern GPUs offer much computing power at a very modest cost. Even though CUDA and other related recent developments are accelerating the use of GPUs for general purpose applications, several challenges still remain in programming the GPUs. Thus, it is clearly desirable to be able to program GPUs using a higher-level interface.

In [47], **Agrawal and Ma** have developed a solution for high-level programming of GPUs. Their proposed solution targets a specific class of applications, which are the data mining and scientific data analysis applications. They exploit the common processing structure, generalized reductions that fit a large number of popular data mining algorithms in the GeForce 8800GTX and 9800GX2 graphics cards. In their proposed solution, the programmers simply need to specify the sequential reduction loop(s) with some additional information about the parameters. Program analysis and code generation is used to map the applications to a GPU. Several additional optimizations (mainly on optimizing memory usage) are also performed to improve the performance.

The evaluation of the system was made by using three popular data mining applications, k-means clustering, EM clustering, and Principal Component Analysis (PCA). The speedup that each of these applications achieved over a sequential CPU version ranges between 20 and 50 and the code automatically generated by the system did not have any noticeable overheads compared to hand written codes.

In [48], **Fang et al.** presented two efficient *Apriori* implementations of Frequent Itemset Mining (FIM) that utilize new-generation graphics processing units (GPUs). Their implementations take advantage of the GPU's massively multi-threaded SIMD (Single Instruction, Multiple Data) architecture. Both implementations employed a bitmap data structure to exploit the GPU's SIMD parallelism and to accelerate the frequency counting operation. One implementation runs entirely on the GPU and eliminates intermediate data transfer between the GPU memory and the CPU memory. The other implementation employs both the GPU and the CPU for processing. It represents itemsets in a trie, and uses the CPU for trie traversing and incremental maintenance. Our preliminary results show that both implementations achieve a speedup of up to two orders of magnitude over optimized CPU *Apriori* implementations on a PC with an NVIDIA GTX 280 GPU and a quad-core CPU.

The GPU-based implementations have larger speedup over the CPU-based ones on the dense dataset than on the sparse dataset. The speedup achieved varied from 1.2 to 130 times faster than the CPU implementation and it depends on the algorithm selection and the input data's nature.

3.5 Support Vector Machines

Support Vector Machines [30] are a powerful machine learning method, providing good generalization performance for a wide range of regression and classification tasks. In SVMs, a training dataset, consisting of pairs of input vectors and desired outputs, is used to model and construct the decision function of the system and, hence, they are considered as an instance of supervised learning. During the training phase, the system identifies the Support Vectors (SVs), which are those data points that can best build a separation model for the classes. Those vectors are then used to predict the class of any future data point during the classification phase.

Papadonikolakis and Bouganis [49, 51] proposed a scalable FPGA architecture for the acceleration of SVM classification, which exploits the device heterogeneity and the dynamic range diversities among the dataset attributes. Furthermore, this work introduced the first FPGA-oriented cascade SVM classifier scheme, which intensified the custom-arithmetic properties of the heterogeneous architecture and boosts the classification performance even more.

The rationale behind the design of the SVM classifier is the exploitation of the parallel computational power offered by the FPGA heterogeneous resources and the high memory bandwidth of the FPGA internal memories in the most efficient way, in order to speed up the decision function. The computation of this function involves matrix-vector operations, which are highly parallelizable. Therefore, the problem can be segmented into smaller ones and parallel units can be instantiated for the processing of each sub-problem.

The targeted device for the proposed architecture was the Altera's Stratix III EP3SE260. The results can be expanded to other targeted devices by changing the resource constraints of the design flow. The architecture is captured in VHDL and the floating-point modules are generated by the Altera tools and the Altera floating-point compiler. The targeted operating frequency of all produced designs ranged between 180-250MHz.

The implementation results demonstrated the efficiency of the heterogeneous architecture, presenting a speed-up factor of 2-3 orders of magnitude, compared to the CPU implementation, while performing much better other proposed FPGA and GPU approaches by more than 7 times.

As it is shown from the previously mentioned related work, the field of data mining is promising in terms of reconfigurable logic implementation. The main reason is that the data mining algorithms can be easily parallelized, a fact that can be fully exploited by the FPGA's characteristics and can lead to significant reduce of the execution time compared to the corresponding software running time. However, as shown before, a few approaches were made on this field and especially on decision tree classification, giving us a motivation to study classification algorithms in this thesis in order to accelerate them.

4. SW ANALYSIS

This section describes WEKA, the open source software java platform that was used in this work and analyzes the basic steps of the BFTree classification algorithm which was the case study in this thesis.

4.1 WEKA tool

Figure 4-1: Weka's startup screen

The Waikato Environment for Knowledge Analysis (Weka) is a comprehensive suite of Java class libraries that implement many state-of-the-art machine learning and data mining algorithms.

Weka is freely available on the World-Wide Web and accompanies a text on data mining [52] which documents and fully explains all the algorithms it contains. Applications written using the Weka class libraries can be run on any computer with a java runtime environment capability; this allows users to apply machine learning techniques to their own data regardless of computer platform.

Tools are provided for pre-processing data, feeding it into a variety of learning schemes, and analyzing the resulting classifiers and their performance. An important resource for navigating through Weka is its java documentation (javadoc), which is automatically generated from the source code. The primary learning methods in Weka are "classifiers", and they induce a rule set or decision tree that models the data. Weka also includes algorithms for learning association rules and clustering data. All implementations support a uniform command-line or a Graphic

User Interface (GUI). A common evaluation module measures the relative performance of several learning algorithms over a given data set.

Tools for pre-processing the data, or "filters," are another important resource. Like the learning schemes, filters have a standardized command-line interface with a set of common command-line options.

The Weka software is written entirely in Java to facilitate the availability of data mining tools regardless of computer platform. The system is, in sum, a suite of Java packages, each documented to provide developers with state-of-the-art facilities.

4.1.1 Javadoc and the class library

One advantage of developing a system in Java is its automatic support for documentation. Descriptions of each of the class libraries are automatically compiled into HTML, providing an invaluable resource for programmers and application developers alike. The Java class libraries are organized into logical packages - directories containing a collection of related classes. These packages provide interfaces to pre-processing routines including feature selection, classifiers for both nominal and numeric learning tasks, meta classifiers for enhancing the performance of classifiers (for example, boosting and bagging), evaluation according to different criteria (for example, accuracy, entropy, root-squared mean error, cost-sensitive classification, etc.) and experimental support for verifying the robustness of models (cross-validation, bias-variance decomposition, and calculation of the margin).

4.1.2 Weka's core

The *core* package contains classes that are accessed from almost every other class in Weka. The most important classes in it are *Attribute*, *Instance*, and *Instances*. An object of class Attribute represents an attribute - it contains the attribute's name, its type, and, in case of a nominal attribute its possible values. An object of class Instance contains the attribute values of a particular instance; and an object of class Instances contains an ordered set of instances—in other words, a dataset. Figure 4.2 shows a simple example of a Weka's dataset, which follows the previously mentioned structure rules. More details about the dataset's structure will be discussed in the next section.

```
@relation weather.symbolic

@attribute outlook {sunny, overcast, rainy}
@attribute temperature {hot, mild, cool}
@attribute humidity {high, normal}
@attribute windy {TRUE, FALSE}
@attribute play {yes, no, maybe}

@data
sunny,hot,high,FALSE,no
sunny,hot,high,TRUE,no
overcast,hot,high,FALSE,yes
rainy,mild,high,FALSE,yes
rainy,cool,normal,FALSE,yes
rainy,cool,normal,TRUE,no
```

Figure 4-2: An example of a Weka's dataset

4.1.2.1 Pre-processing utilities

Real databases invariably contain large quantities of information that must be greatly reduced before processing. Most machine learning schemes only work on comparatively impoverished two-dimensional "flat-file" views of the data. Thus, considerable manipulation of a database is invariably necessary before any information can be processed by WEKA. This effort ranges from performing SQL queries on relational databases, through writing macros for spreadsheets, to the invocation of pattern matching scripts to process text files.

The dataset resulting from these operations is then processed on WEKA as follows (fig. 4-3):

• A data file is selected from the **Open File** menu

• Statistical characteristics of the data are viewed using XLISPSTAT [53]

• Important attributes of the data are selected

• Aggregates of existing attributes are created using the spreadsheet

• A machine learning scheme is selected from the menu (Classify, Cluster and Associate tabs from menu)

• The results are viewed as trees, text or three dimensional plots (Visualize tab)

• Attribute/aggregate selections are revised

• The scheme is re-run on the revised data

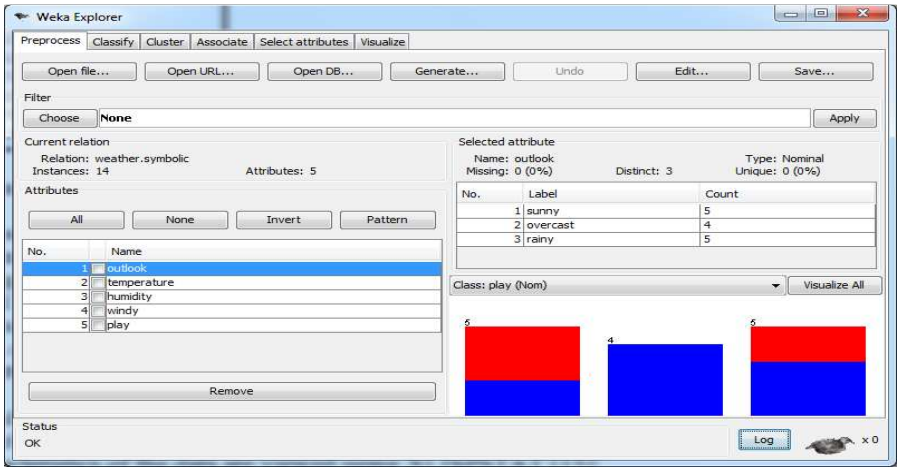

Figure 4-3: An example of a Weka's dataset

In order to maintain format independence, data is converted to an intermediate representation called ARFF (Attribute Relation File Format). Figure 4-2 shows an example ARFF file containing data describing instances of weather conditions and whether or not to play golf. ARFF files contain blocks describing relations and their attributes, together with all the instances of the relation - and there are often very many of these, as the instances number can reach up to a few thousand. They are stored as plain text for ease of manipulation. Relations are simply a single word or string naming the concept to be learned. Each attribute has a name, a data type (which must be one of enumerated, real or integer) and a value range (enumerations for nominal data, intervals for numeric data). The instances of the relation simplify interaction with spreadsheets and databases. Missing or unknown values are specified by the '?' character.

When a machine learning scheme is invoked, the data set is converted to an input form, appropriate for that scheme by using a customized filter. The input as well as the output is converted individually for each scheme, so that it is not necessary to rewrite a machine learning scheme in order to incorporate it into the workbench. These applicable filters are categorized depending on the target structure, either in the instance or in the attribute. An indicative example of the filters that can be used for the conversion of the initial input is shown below:

Attribute Filters:

- Add
- Discretize
- MergeTwoValues
- RemoveUseless
- Reorder
- Standardize
- StringToNominal
- SwapValues

Instance filters:

- Normalize
- Randomize
- RemoveFrequentValues
- RemoveRange
- Resample

In addition to the filters that are customized for machine learning schemes, a range of filters is available for converting new files to the ARFF format. For many applications, it is necessary to construct new ARFF files from existing ones. Many of the schemes produce results which inform the user that certain attributes do not contribute towards classification, and a user may wish to remove these irrelevant attributes.

A vital requirement, which was discovered only after examining very large data sets [54], is the ability to compute aggregates of attributes. These often lead to a far more satisfactory classification than that obtained with the original attributes. They are typically averages of existing attributes, or differences from an attribute's mean value. They can only sensibly be suggested by people who understand the data (including where and how it came into existence). It is, therefore, imperative to provide within the environment tools for data analysis such as spreadsheets, and basic statistical display tools. WEKA has a built-in spreadsheet capable of generating new ARFF files from existing ones; moreover, it invokes a statistical package (XLISPSTAT [53]) to display histograms of attribute values, scatterplots, boxplots, and three-dimensional views of the data.

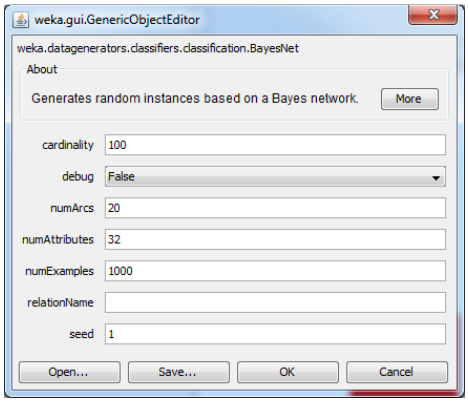

Figure 4-4: Data Generator Tool

Another important feature that Weka offers is the ability to generate a random ARFF file from the DataGenerator tool. Thus, the user can easily create his own input dataset by choosing among several distributions such as Agrawal, BayesNet, RandomRBF and RDG1 by simply selecting the number of the desirable instances, the number of the attributes as well as the possible values of each attribute (figure 4-4). By using this feature, it was possible to generate several testbenches that gave us the ability to test the algorithm's implementation in extreme cases without having to search the web for the suitable input.

4.1.2.2 Post-processing utilities

In an environment where many different learning schemes are available, it is important to be able to evaluate and compare the results produced by each one. WEKA provides cross-validation studies to be performed, and incorporates a new method for comparing classifications and rule sets [55]. This method evaluates classifications by analyzing rule sets geometrically. Rules are represented as objects in *n*-dimensional space, and the similarity of classes is computed from the overlap of the geometric class descriptions.

The system produces a correlation matrix that indicates the degree of similarity between pairs of classes. For applications with only a few attributes, the way in which different rule sets cover the data they were inferred from can be viewed and compared.

4.2 Supported classifier trees

The Weka tool supports a vast amount of classification methods and especially tree classifiers, which are the case study of this MSc thesis. The available classification trees are the following:

- ADTree
- BFTree
- DecisionStump
- Id3
- J48
- LADTree
- LMT
- NBTree
- LMT
- RandomFOrest
- RandomTree
- REPTree
- Custom User Classifier

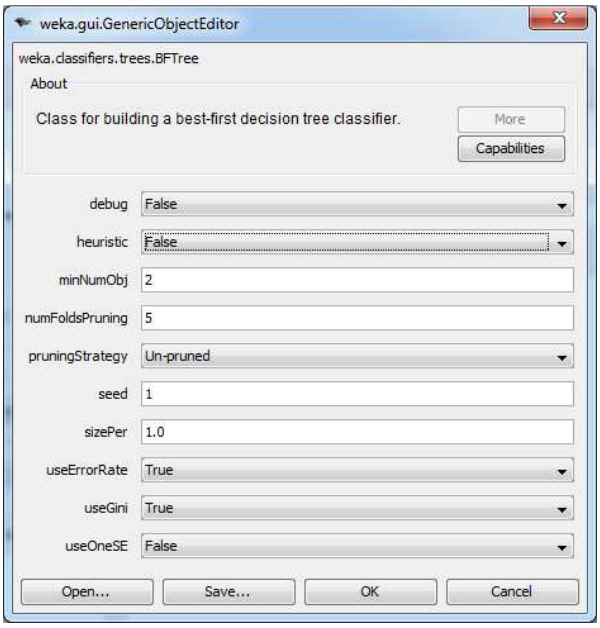

Figure 4-5: The available options for tree classifiers

In Figure 4-5 there is a class for building decision tree classifier. This class uses binary split for both nominal and numeric attributes. For missing values, the method of “fractional” instances is used.

The possible options for the classifier are summarized as follows:

- Debug: If set to true, classifier may output additional info to the console.
- Heuristic: If heuristic search is used for binary split for nominal attributes
- MinNumObj: Set minimal number of instances at the terminal nodes. Thus, if for an example of MinNumObj = 2, if the remaining instances to be examined are equal to 2, no further algorithmic calculations will take place and the remaining dataset is considered to be a leaf node.
- NumFoldsPruning: Number of folds in internal cross-validation.
- PruningStrategy: Sets the pruning strategy. The available options are an unpruned, post-pruned or a pre-pruned tree.
- Seed: The random number seed to be used.
- SizePer: The percentage of the training set size (0-1, 0 not included).
- UseErrorRate: If error rate is used as error estimate. If not, root mean squared error is used.
- UseGini: If true the Gini index is used for splitting criterion, otherwise the information gain is used.
- UseOneSE: Use the 1SE rule to make a pruning decision.

Figure 4-6 shows a typical run of a tree classifier and its output results that are printed in the java console. The output starts with the description of the examined instances; the testcases’s name, the number of the examined instances and the existing attributes. Afterwards, the Decision Tree is printed with the splitting attributes and their corresponding splitting value of each tree node. The resulted tree is better shown in 2d depiction in figure 4-7. Finally, the time to build the model is shown, as well as some statistics and error analyses of the input data set.

```
=== Run information ===

Scheme:       weka.classifiers.trees.BFTree -S 1 -M 2 -N 5 -H -C 1.0 -P UNPRUNED
Relation:     weather.symbolic
Instances:    14
Attributes:   5
              outlook
              temperature
              humidity
              windy
              play
Test mode:    evaluate on training data

=== Classifier model (full training set) ===

Best-First Decision Tree

outlook=(overcast): yes(4.0/0.0)
outlook!=(overcast)
|  humidity=(normal)
|  |  windy=(FALSE): yes(3.0/0.0)
|  |  windy!=(FALSE): yes(1.0/1.0)
|  humidity!=(normal)
|  |  outlook=(rainy): yes(1.0/1.0)
|  |  outlook!=(rainy): no(3.0/0.0)

Size of the Tree: 9

Number of Leaf Nodes: 5

Time taken to build model: 0.01 seconds

=== Evaluation on training set ===
=== Summary ===

Correctly Classified Instances          12               85.7143 %
Incorrectly Classified Instances        2               14.2857 %
Kappa statistic                          0.6585
Mean absolute error                      0.0952
Root mean squared error                  0.2182
Relative absolute error                 28.8136 %
Root relative squared error             55.389  %
Total Number of Instances               14
```

Figure 4-6: A typical run of a tree classifier

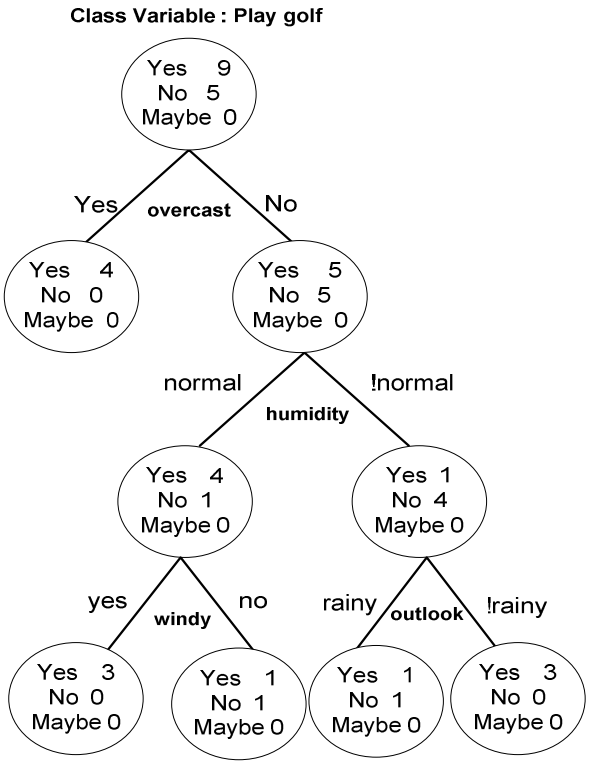

Figure 4-7: The resulted tree classifier of the example

4.3 BFTree decision tree learning

In this MSc thesis, our case study is the Best First Tree (BFTree) algorithm, which is one of the previously mentioned classification methods for decision trees. In the following sections, the basic terminology of the decision trees' characteristics will be described, as long as the basic steps of the BFTree algorithm, some of which were selected to be implemented in reconfigurable logic (chapter 5).

Trees generated by best-first decision tree learning have all properties described in the previous chapter. The only difference is that, standard decision tree learning expands nodes in depth-first order, while best-first decision tree learning expands the "best" node first. Standard decision tree learning and best-first decision tree learning generate the same fully-expanded tree for a given data. However, if the number of expansions is specified in advance, the generated trees are different in most cases.

For example, Figure 4-8 shows a hypothetical standard decision tree and a hypothetical best-first decision tree with three expansions on the same data. The first tree in the figure is the fully expanded tree generated by best-first decision tree learning and the second tree is the fully-

expanded tree generated by standard decision tree learning. In this example, considering the fully expanded best-first decision tree the benefit of expanding node N2 is greater than the benefit of expanding N3.

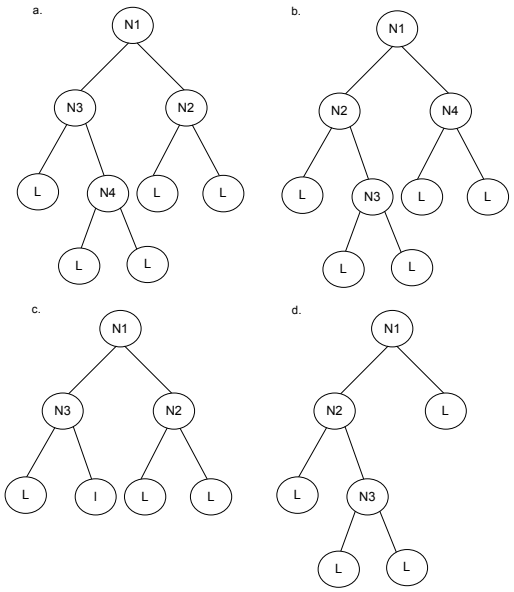

Figure 4-8: (a) The fully-expanded best-first decision tree; (b) the fully-expanded standard decision tree; (c) the best-first decision tree with three expansions from (a); (d) the standard decision tree with three expansions from (b)

4.3.1 Splitting criteria

In order to find the "best" node to split at each step of a best-first decision tree, splitting criteria must be addressed. There are many criteria for decision trees and two of them are most widely used, the information and the Gini index. For example, the information is used in ID3 [24] and C4.5 [26] and the Gini index is used in the CART system [27].

The Best-first decision trees can also use these two criteria. Splitting criteria are designed to measure node impurity. The node impurity is based on the distribution of classes. The main objective of decision tree learning is to obtain accurate and small models. Thus, when splitting a node, a pure successor node should be found as early as possible. In other words, the goal of the splitting is to find the maximal decrease of impurity at each node. The decrease of impurity is calculated by subtracting the impurity values of successor nodes from the impurity of the node. When the subtraction is performed, the impurity values of the successor nodes are weighted by the size of each node: the number instances reaching each node. If the splitting criterion is the information, the decrease in impurity is measured by the information gain. Similarly, if the splitting criterion is the Gini index, the decrease in impurity is measured by the Gini gain.

Gini index is another criterion to measure the impurity of a node, which is used in the CART system (Breiman et al., 1984). If p_i stands for the probability that an instance is in class i and p_j is the probability that the instance is in class j, the Gini index has the form (Breiman et al., 1984):

$$gini(p_1, p_2, \dots, p_n) = \sum_{j \neq i} p_i p_j$$

The Gini index has an interesting interpretation in terms of sample variances (Breiman et al., 1984). If all instances at a node that are in class j are assigned the value 1 and the other instances are assigned the value 0, the sample variance of these values is $p_j(1 - p_j)$. Repeating this for all classes and summing the variances, as the sum of the p_j over all classes is 1, it can be shown that the Gini index can be written as follows.

$$gini(p_1, p_2, \dots, p_n) = 1 - \sum_j p_j^2$$

Note: When only two classes are presented, the Gini index can be simplified to $2p_1p_2$.

4.3.2 Splitting rules

The goal of decision tree learning is to find accurate and small models. To get the smallest trees, heuristically, all attributes need to be tested and then, the purest nodes to split at each step should be chosen. In other words, the goal of splitting rules is to find the split value and attribute which maximally reduces the impurity. Thus, if the information or the Gini index is used as the splitting criterion, the task of splitting rules is to find the split which leads to maximal information gain or Gini gain. In fact, finding the maximal Gini gain or information gain for a split at a node is to find the minimal values of the weighted sum of the information values or the Gini index values of its successor nodes.

This section describes splitting rules for numeric attributes and nominal attributes in best-first decision trees. For numeric attributes, the splitting rule is the same as the one used in C4.5 [26] and the CART system [27]. For nominal attributes, in multi-class problems both exhaustive search and heuristic search are discussed. The exhaustive search used is the same as in the CART system. The computation time of the exhaustive search is exponential in the number of values of a nominal attribute. Heuristic search can reduce the search time to linear. It is obvious

that for an attribute that has many values heuristic search is the better choice because it can reduce computation time significantly.

4.3.2.1 Numeric attributes

A numeric attribute can be viewed as a sequence of ordered values. Thus, there can be many potential split points to divide training instances into two subsets, one for each pair of adjacent values. The reduction of impurity must be computed for each split point. When choosing split points, all the numeric values of training instances for the attribute concerned are first sorted in ascending order. Then the split points are set to the midpoints of two different adjacent values. The goal of the splitting rule for the numeric attribute is to find the midpoint that leads to the maximal reduction of impurity. No further details on numeric attributes will be discussed, as our case study is restricted to the nominal attributes. The summary of the assumptions and the conventions made in this MSc thesis are summarized in the last section of this chapter.

4.3.2.2 Nominal attributes

The splitting rule for nominal attributes is quite different from the one for numeric attributes. The goal of the splitting rule for a nominal attribute is to find a subset of attribute values that can maximally reduce impurity. When separating instances into two branches, if the value of an instance at the attribute is in the subset of values, the instance is placed into the left subset. Otherwise, it is placed into the right subset. Suppose that the nominal attribute A takes a set of values $\{a_1, a_2, \ldots, a_n\}$ at a node, where n is the number of attribute values. The goal of the splitting rule is to search for a subset of value:

$$A^* = \{a_{i1}, \ldots\} \subset \{a_1, a_{2,\ldots,} a_n\},$$

where the split based on this subset can maximally reduce impurity. Note that this subset can be more than one due to ties. To find such subset, 2^{n-1}-1 subsets need to be looked in the exhaustive search case as there is no difference in placing a set of values into the left branch or the right branch in a binary tree. The rest of this section introduces two search methods that can reduce the search space to linear. One is for two-class problems and the other is for multi-class problems.

4.3.3 The two-class problem

In a two-class problem, Breiman et al. (1984) introduced a search method that can reduce the binary search space from 2^{n-1}-1 to n-1. The principle is as follows. If the probability of being in

class 1 for a given single value a_i at a node is denoted $p(1|x = a_i)$, for all single attribute values we sorted their class probabilities for class 1 as:

$$p(1|x = a_{i1}) \leq p(1|x = a_{i2} \leq \cdots \leq p(1|x = a_{in})$$

Then, it can be shown that one of the subsets $\{a_{i1}, \ldots, a_{im}\}$ (m=1,. . ., n − 1) is the subset that maximally reduces the impurity if the Gini index is used as impurity measure. This strategy is also used for the information gain. The underlying idea is that the best split should put all those attribute values leading to high probabilities in class 1 into one branch and the attribute values leading to low probabilities in class 1 into another branch.

4.3.4 The multi-class problem

In a multi-class problem, Coppersmith et al [56] introduced a heuristic search method that achieves a compromise between reduction of impurity and search speed. For the optimal reduction of impurity, the method searches for a partition based on a separating hyperplane in the class probability space. For search speed, it assigns a scalar value to each attribute value and forms a sequence of sorted attribute values to split.

According to [56], this procedure usually finds the splitting subset which achieves an optimal split or a split very near to the optimal split. However, it only requires n-1 total impurity evaluation time instead of 2^{n-1}-1.

4.3.5 The selection of attributes

The splitting criteria and splitting rules have been described above. The next thing is to find the “best” attribute among all attributes to split on at a node. The “best” attribute is the attribute that leads to the split of maximal reduction of impurity. Note that this attribute is sometimes not unique.

4.4 Best-first decision trees

Like standard decision trees, best-first decision trees are constructed in the divide and conquer fashion. Each non terminal node in a best-first decision tree tests an attribute and each terminal node is assigned a classification. During the process of construction, three important things must be considered.

The first one is to find the best attribute to split on at each node. The second one is to find which node in the node list (i.e. all nodes that are candidates for splitting) is to be expanded next. The third one is to make the decision when to stop growing trees. The selection of the best attribute

and the corresponding splitting value at a node has been discussed in Section 4.3: the attribute that leads to the maximal reduction of impurity is chosen to split on among all attributes.

For a numeric attribute, the splitting point that achieves maximal reduction of impurity is the splitting point of the attribute, and the corresponding reduction of impurity is the reduction of impurity of the attribute. For a nominal attribute, the subset of attribute values that leads to the maximal reduction of impurity is the splitting subset of the attribute, and the corresponding reduction of impurity is the reduction of impurity of the attribute. Best-first decision tree learning chooses the "best" node to split at each step. The "best" node is the node that has the maximal reduction of impurity among all nodes in the node list. This node can be any one in the list while it is always the same one in standard decision tree learning (as determined by the depth-first search order).

To save searching time, it is a good idea to sort all nodes in the list in descending order according to Gini gain or information gain (i.e. to keep a priority queue). After sorting, the first node in the list is always the one to be expanded next. If the reduction of impurity of the first node is zero, the reduction of all nodes in the list is zero and all nodes cannot be split any more.

Regarding the stopping criteria, standard decision tree learning stops expanding a tree when all nodes are pure or the impurity of all nodes cannot be reduced by further splitting. Sometimes a minimal number of instances is required. However, besides these stopping criteria, in best-first decision tree learning, a fixed number of expansions could be specified. A tree stops expanding when a fixed number of expansions is reached. In standard decision tree learning, specifying a number of expansions is not meaningful as the order of node splitting is fixed. This stopping criterion enables the investigation of pre-pruning and post-pruning methods by choosing the fixed number of expansions based on cross-validation.

4.4.1 The best-first decision tree learning algorithm

Figure 4.9 presents the best-first decision tree learning algorithm examined in this thesis. Given a set of training instances E, its set of attributes A, the fixed number of expansions N and the fixed minimal number of instances at a terminal node M, the algorithm can be divided into two stages.

In the **first stage**, the algorithm starts at the root node RN and finds the best splitting attribute A_b in A according to the reduction of impurity. E and A_b are kept in RN. Then, RN is added into the empty node list NL.

In the **second stage**, the first node FN in NL, and its corresponding best splitting attribute A_b and the instances reaching the node E are retrieved first. If the reduction of impurity of FN is 0 or N is reached, all nodes in the list NL are made into terminal nodes and the process of constructing the best-first decision tree is finished. Otherwise, if the split of FN on A_b would lead to a successor node with fewer than M instances, FN should not be split and it is removed from NL. Then, the algorithm executes this stage again with the new node list NL. If the split does not leads to this circumstance, FN is split into two successor nodes SN_1 and SN_2 (i.e. branches) based on its best splitting attribute A_b, and its training instances E are separated into two corresponding subsets E_1 and E_2, one for each branch.

Then, the best reduction of impurity for SN_1 and SN_2, and the corresponding best splitting attributes A_{b1} and A_{b2} are calculated. In the next step SN_1 with A_{b1} and E_1, and SN_2 with A_{b2} and E_2 are added into NL according to the reduction of impurity (i.e. NL is kept in sorted order). The number of expansions of the tree is incremented by one. Next, FN is removed from NL. Finally, this stage is repeated with the new node list NL (figure 4.9).

When building decision trees, for a numeric attribute, it is a good idea to sort the training instances by the values of the attribute at the root node and then every descendant node can use the sort order from its parent node. The reason is that, for a very large dataset, if the sorting of instances takes place at each node, the sorting time is very expensive. To achieve the goal, the only thing that needs to be done is to keep the sorted indexes of the parent node, and the successor nodes can then derive the indexes from the parent node. The time of the derivation is linear in the number of instances while the time of re-sorting instances is log-linear. For a nominal attribute, as mentioned before, if exhaustive search is used, the computation time is exponential in the number of values of the attribute. If heuristic search is used, the computation time grows linearly in the number of the attribute values.

```
function BFTree (A: a set of attributes,
                 E: the training instances,
                 N: the number of expansions,
                 M: the minimal number of instances at a terminal node
) return a decision tree
    begin
        If E is empty, return failure;
        Calculate the reduction of impurity for each attribute
              in A on E at the root node RN;
        Find the best attribute A_b in A;
        Initialise an empty list NL to store nodes;
        Add RN (with E and A_b) into NL;
        expandTree(NL, N, M);
        return a tree with the root RN;
    end


expandTree(NL, N, M)
    begin
        If NL is empty, return;
        Get the first node FN from NL;
        Retrieve training instances E and the best splitting attribute A_b of FN;
        If E is empty, return failure;
        If the reduction of impurity of FN is 0 or N is reached,
            Make all nodes in NL into terminal nodes;
            return;
        If the split of FN on A_b would result in a successor node
                with less than M instances,
            Make FN into the terminal node;
            Remove FN from NL;
            expandTree(NL, N, M);
        Let SN_1 and SN_2 be the successor nodes generated by
            splitting FN on A_b on E;
        Increment the number of expansions by one;
        Let E_1 and E_2 be the subsets of instances corresponding to
            SN_1 and SN_2;
        Find the corresponding best attributes A_b1 for SN_1;
        Find the corresponding best attributes A_b2 for SN_2;
        Put SN_1 (with E_1 and A_b1) and SN_2 (with E_2 and A_b2)
            into NL according to the reduction of impurity;
        Remove FN from NL;
        expandTree(NL, N, M);
    end
```

Figure 4-9: BFTree's algorithm basic steps in pseudocode

4.4.2 Missing values

Missing values are endemic in real world datasets for reasons such as malfunctioning equipment or missing measurements. One way of dealing with this problem is to simply treat them as another possible value of the attribute, which is most applicable if the fact that a value is missing plays a significant role in the decision. However, often the fact is that a value missing has no special significance. Under these circumstances, instances with missing values are similar to other instances. Witten and Frank [57] outlined a solution by splitting instances into pieces, using a numeric weighting scheme.

4.4.3 Pruning

As in standard decision trees, using an unpruned decision tree for classification potentially overfits the training data because of noise and variability in the data. Figure 4.10 shows the number of expansions and the corresponding accuracy based on ten-times ten-fold cross-validation for best-first decision tree learning on the iris dataset. Note that the minimal number of instances at a terminal node has been set to one in this case. The figure illustrates that, first, the accuracy increases when the expansions start. Then, the accuracy peaks when the number of expansions reaches three and further expansions do not improve performance. Thus, they should not be performed (i.e. pre-pruning) or they should be deleted if they have already performed (i.e. post-pruning).

The goal of pruning is to only retain those parts that truly reflect the underlying information in the data and remove the others. The pruning methods that are supported in the BFTree implementation are, namely, best-first-based pre-pruning and post-pruning, which try to find the appropriate number of expansions for best-first decision trees. The tree size is decided according to the error estimate obtained from an (internal) cross-validation.

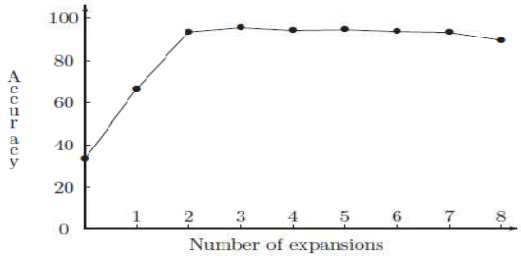

Figure 4-10: The accuracy for each number of expansions of best-first decision tree learning on an example dataset.

4.4.3.1 Best-first-based pre-pruning

Like in standard decision trees, pre-pruning in best-first decision trees stops expanding a tree early when further splitting step appears to increase error estimate (i.e. the error rate or the root mean squared error in this thesis). The best-first-based pre-pruning method considered here is based on cross-validation. To this end, the trees for small training folds are constructed in a parallel fashion (e.g. ten trees for a ten-fold cross-validation). For each number of expansions, the average error estimate is calculated based on the temporary trees in all folds. Best-first-based pre-pruning stops growing the trees when further splitting increases the average error estimate and the previous number of expansion is chosen as the final number of expansions.

Then, the final tree is built according to the chosen number of expansions on all the data. In all folds, each best-first tree expands a node at a time. Note that in each fold, when expanding a tree, no tree regeneration is required. The only thing that has to be done is to keep the previous tree and select the next "best" node from the node list to split. This can significantly save computation time. Like in pre-pruning methods for standard decision trees, the pre-pruning algorithm also suffers from the problem that the process may stop too early, and further splitting may decrease the error estimate again.

4.4.3.2 Best-first-based post-pruning

For best-first-based post-pruning, trees in all training folds are also constructed in a parallel fashion. For each number of expansions, the average error estimate is calculated based on the temporary trees in all folds. This step is repeated until the trees cannot be expanded any more. Then, a sequence of the number of expansions and their corresponding error estimates based on the cross-validation can be calculated. The number of expansions whose average error estimate is minimal is chosen as the final number of expansions. The final tree is then built according to the chosen number of expansions on all the data. Like in best-first-based pre-pruning, in each fold, when expanding a tree, only the previous tree has to be kept and then the next "best" node from the node list to split is selected in order to save computation time. Best-first-based post-pruning does not suffer from the problem described in best-first-based pre-pruning as it considers all possible expansions regardless whether an expansion increases the error estimate or not.

4.4.4 Complexity of best-first decision tree induction

The best-first decision tree learning algorithm and the two new pruning algorithms have been discussed. Now, let us consider their computational complexity using the big O notation. The method used here is similar to the one used in Witten and Frank [57]: O(n) stands for a quantity that grows at most linearly with n and $O(n^2)$ stands for a quantity that grows at most quadratically with n. Assume that a tree is built on a dataset which has n training instances and m attributes, and the depth of the tree is on the order of O(logn). In the building process, each node requires effort that is linear in the number of instances, as all instances need to be considered at each level, the amount of work for one attribute is O(nlogn). Also, at each node, all attributes are considered, so the time complexity of building the tree is O(mn x logn).

There is a difference between the time complexity of numeric attributes and nominal attributes. In the case of all numeric attributes, the complexity of sorting instances for an attribute at the root node is O(nlogn). If descendant nodes derive the sorting order from the root node, the complexity is still O(mn x logn).

In the case of all nominal attributes, at each node the best subset of an attribute values to split has to be found. Assume the number of values of an attribute is k. In the exhaustive search case, the computation time for the best subset search for an attribute is $2^{k-1} - 1$ and in big O notation this is $O(2^k)$. As the tree is binary, the number of nodes is 2n − 1 at most and in big O notation this is O(n), so the worst-case time complexity for a nominal attribute is $O(2^k n)$. Thus, the time complexity for building the tree is $O(m2^k n)$. In the heuristic search case, as the time required for the best subset search for an attribute becomes O(k), the time complexity for building the tree is O(kmn).

Let F be the number of folds. Let O(X) (where X refers to one of the three possible time complexities described above) be the time complexity for building a tree. If best-first-based pre-pruning or post-pruning are applied, as in each fold a tree needs to be built, the time complexity for building trees in all folds would be FX and in big O notation it is O(X) because F is a small constant compared to n. In each fold, an error estimate is computed for every node. In the worst case, the number of nodes is 2n − 1 and in big O notation this is O(n). Thus the time complexity for the estimates for all folds is F(2n − 1) and in big O notation this is O(n). Obviously, there is no difference between best-first-based pre-pruning and post-pruning because in the worst case best-first-based pre-pruning needs to fully expand the tree. Thus, the computation time of the whole process is O(X)+O(n), which is O(X).

4.5 Thesis' implementation conventions

After profiling the Java Weka code for a BF-Tree run, the timing results showed that the most time consuming operation was the calculation of the splitting criterion for the DT tree nodes. These calculations are shown in figure 4.9 and they approximately consume the 99% of the total BF-Tree algorithm runtime. Thus, the reconfigurable system implements the previous functions, while the rest of the BF-Tree algorithm will still run on software.

The conventions made in this MSc thesis are the following:

- While the processing of the nominal attributes demand $O(2^k n)$ time and the numeric attributes O(nlogn) time, it is obvious that the calculation of the splitting criteria for the nominal attributes is much more time consuming. Thus, the hardware implementation is designed to examine the nominal attributes, while the numeric ones are left on software. For future work and with a few changes in the architecture, both nominal and numeric attributes can be examined by the reconfigurable system.
- The goal of this dissertation is to compare the executions times of the original software and the corresponding hardware implementation. As a result, some available features of the BF-Tree construction method were not taken into consideration, although they can be considered for future work in the initial HW implementation. To be more specific:
- The pruning methods (post and pre-pruning) were not implemented in reconfigurable logic, as the software can easily process the input data (for pre-pruning) or the output decision tree (for the post pruning feature) without any significant overhead added.
- The examination of the nominal attributes follows the exhaustive search rules in order to have a more precise decision tree construction. Thus, the previously referred heuristic functions were not implemented.
- The metric value to measure the impurity decrease of a specific node is considered to be the gini gain. The information gain can also be implemented instead with minimal alterations to the original implemented architecture.

5. Architecture

This section presents the FPGA-based first architecture of the Best First Tree construction method. The architecture was implemented on a Xilinx Virtex 5 FPGA Family Member Device. The goal of the system is to compute the best (minimum) Gini gain for all the possible values of the examined attributes in order to find the N^{ary} splitting point and the splitting value of each tree node. The proposed system also implements a counter pattern architecture which computes the frequency of the instances according to their class value that is used for the Gini gain computation.

5.1 First system's architecture

Figure 5-1 shows the overall proposed architecture of a single BF-Tree construction system, which consists of three subsystems.

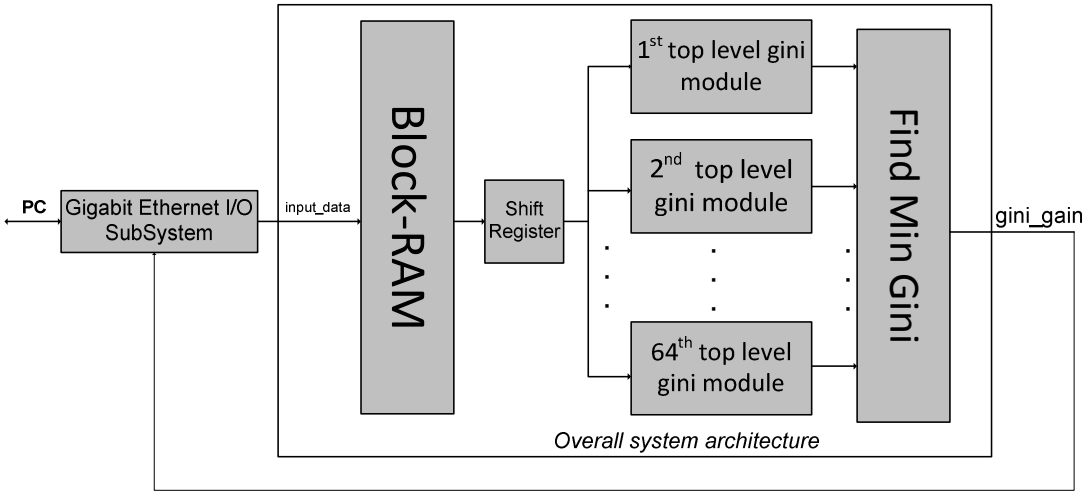

Figure 5-1: The architecture of the FPGA-based BF-Tree construction system

5.1.1 UDP/IP Core

The first subsystem is an efficient UDP/IP core for PC-FPGA communication. The Internet Protocol version 4 (IPv4) is the most widely used Internet Layer protocol and together with IPv6 is at the core of standard-based internetworking methods of the Internet. Even in 2011, IPv4 is still by far the most widely deployed Internet Layer protocol, as IPv6 deployment is still in its infancy. The combination of IPv4 with the User Datagram Protocol (UDP), represents the optimal solution in terms of hardware resource requirements for data transmission between a host PC and an FPGA board. Because of the comparatively small protocol overhead, UDP

offers high transmission rates while—at the same time—requiring low programming overhead on the PC side. Also, the programming interfaces for UDP are well documented and readily available for all common operating systems.

Most modern FPGAs contain EMAC (Ethernet Media Access Controller) blocks that offer direct access to external physical layer (PHY) devices on the board. A PHY is required for the EMAC to connect to an external device, for example a network, a host PC, or another FPGA. For example, one can deploy the Xilinx Core generator to configure and generate EMAC wrapper files that contain a user configurable Ethernet MAC physical interface, e.g., MII, GMII and SGMII in our case, and to instantiate RocketIO serial transceivers, clock buffers, DCMs and generally all the necessary components. Xilinx also provides an optimized clocking scheme for the physical interface as well as a simple FIFO-loopback example design which is connected to the EMAC client interface.

Although the EMAC wrapper files greatly simplify the usage of the EMAC, extra logic is required to create packets that comply with Transport Layer protocols and that will be accepted by the Linux kernel, which is necessary for the intercommunication between the FPGA and the host PC. The Packet Transmitter Unit (PTU) can be connected directly to the EMAC client interface, and also it allows the encapsulation of UDP packets within IPv4 packets (UDP/IP) by using a very small fraction of hardware resources. On an average-sized FPGA like the Virtex 5 LX110T, the UDP/IP core occupies less than 1% of the available slices and can operate at frequency exceeding 125MHz, which is a prerequisite for achieving Gigabit speed. The UDP/IP core is presented in [58] and it is available as open source code for download at: http://wwwkramer.in.tum.de/exelixis/countIPv4.php. The implemented UDP achieves an approximate measured 112 MB/s reception and sending rate.

The UDP/IP core is used for the FPGA Block RAMs' initialization with the input dataset. The convention made for this architecture is that the number of instances of each dataset is restricted by the FPGA's BRAM capacity. The memory's structure (figure 5-2) is determined by the following rules:

- Each examined instance consists of a maximum number of 31 nominal attributes plus a class attribute, making a total of 32 attributes per instance.
- Each nominal attribute can take up to 64 possible values.
- The class attribute can take up to 8 possible values.

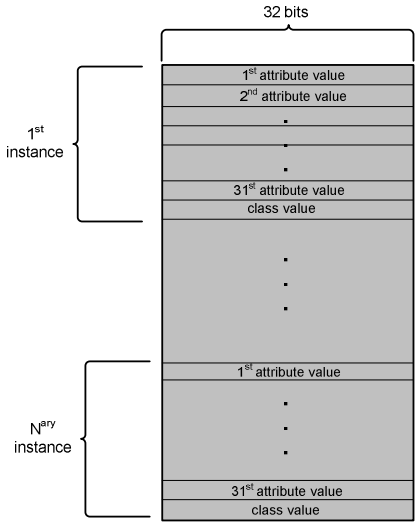

Figure 5-2: The Block RAM's structure

5.1.3 Top level gini module

The second subsystem counts the items' frequencies and calculates the Gini gain for each one of the attributes' values. The total architecture consists of 64 parallel machines that compute the Gini gain for a given attribute value, in order to increase the throughput of the system and to fully exploit the capabilities of the FPGA device. Thus, in this architecture the parallelization is accomplished in the examined values of a given attribute, as 64 parallel machines produce simultaneously the gini gains of the 2^{64} maximum possible attributes' value combinations.

The Top level gini module subsystem, as shown in Figure 5-3, consists of two main sub-modules, the counter pattern module and the Gini Gain module. It takes as input the instances value (attribute value + class value), the number of instances that will be examined and it outputs the gini gain calculated by each attribute value combination.

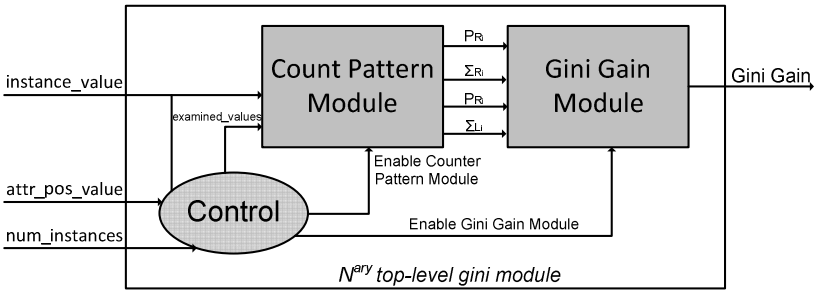

Figure 5-3: The implemented architecture for the Gini module

5.1.4 Counter pattern module

The counter pattern module, as shown in Figure 5-4, computes the instances' frequencies in the input dataset. It takes as input the instances' attribute value as well as the corresponding class value. The control unit, Figure 5-3, takes as input the number of the possible values of the examined attribute and it outputs all the possible combinations of the attribute value that are equal to $2^{\text{possible values}}$.

A simple way to calculate all the possible values' combinations of a given attribute is to initialize a 64-bit counter (the maximum number of values per attribute) with ones and each time the counter is decremented by one until the zero value is reached.

For example, in figure 5-4 the control unit has produced a combination of the attribute values 1, 4, 5, which will then be compared with the instance's attribute value.

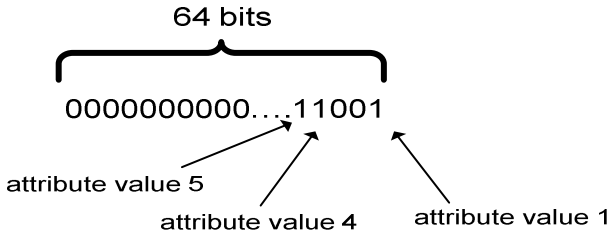

Figure 5-4: An example of the control unit output

In case the outcome of the logic AND between the instances' attribute values and the examined attribute values' combination is greater than zero, the R_i class value memory, Figure 5-5, is written in one of the eight positions that the class value dictates. In the other case, the L_i class value memory is written accordingly. Hence, in each memory position, the written data are equal to the number of times the class value exists in all instances. After all the input data are read – the total number of instances is equal to the examined instances (control unit) – the values of each memory are added and multiplied with each other in order to calculate the final output of the module, according to eq. (2) – (5) :

$$\sum_{i=0}^{i<num\_classes} R_i \ (2), \quad \sum_{i=0}^{i<num\_classes} L_i \ (3), \quad \prod_{i=0}^{i<num\_classes} R_i \ (4), \quad \prod_{i=0}^{i<num\_classes} L_i \ (5)$$

.

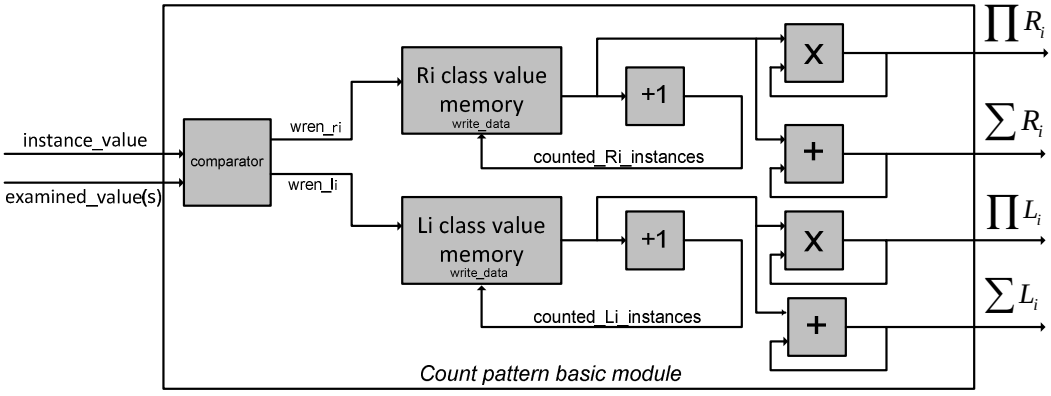

Figure 5-5: The implemented sub-system for the Counter pattern module

As shown in figure 5-6, the R_i memory (and the L_i correspondingly) holds the values of the R_i times that the attribute's examined value belongs to the examined values that came from the control unit.

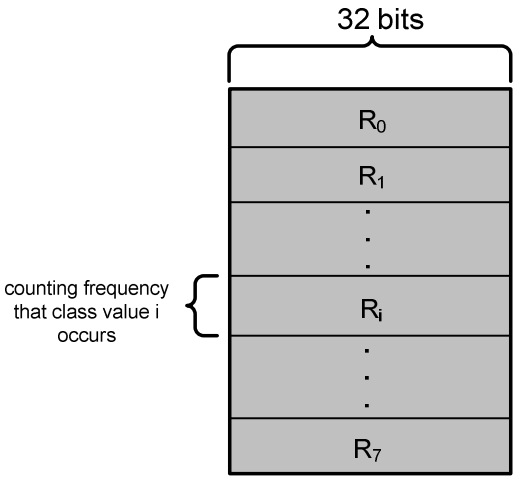

Figure 5-6: Structure of the Ri memory

5.1.5 Gini gain module

The gini gain module, Figure 5-7, is responsible for calculating the gini gain for each sequence of the corresponding four outputs (Equations (2)-(5)) from the counter pattern module. The equation according to which the gini gain is calculated is given in Eq. (6). In order to simplify the mathematical operations, the BFTree's initial equation (section 4.3.1) was modified, so as the minimum gini gain is kept each time, while in the software version, the maximum gini gain is the number that gives the splitting criterion of each tree node.

$$Gini\ Gain = \frac{\prod_{\substack{i=0 \\ i<num\_classes}} R_i}{\sum_{\substack{i=0 \\ i<num\_classes}} R_i} + \frac{\prod_{\substack{i=0 \\ i<num\_classes}} L_i}{\sum_{\substack{i=0 \\ i<num\_classes}} L_i} \qquad (6)$$

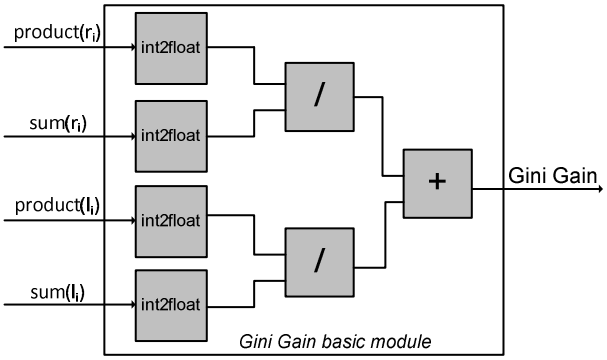

Figure 5-7: The Gini gain module's architecture

5.1.6 Find mini gini module

The final subsystem, Find Mini Gini module, Figure 5-8, is a tree of comparators that takes as input the Gini gains computed by the Gini module and finds the best Gini index for the N^{ary} node's splitting point. As referred above, the best gini gain of this implementation is the minimum calculated number, except for zero value. Due to the 64 parallel machines, the number of comparators that are required in order to find the minimum number of the 64 gini gains that occur from the corresponding calculated products and sums is equal to 64 – 1= 63.

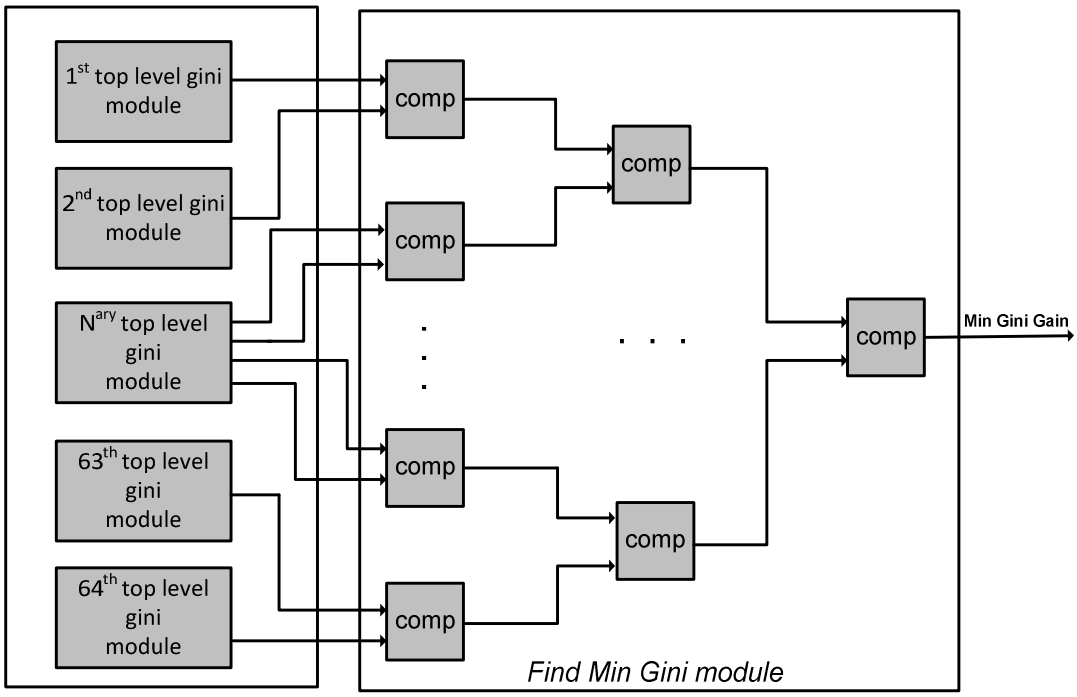

Figure 5-8: The Find Min Gini module's architecture

5.1.7 Implementation details

The mathematical operations made in this implementation were made with the FPGA's Digital Signal Processors (DSPs).

A DSP is an onboard processing unit that provides ultra-fast instruction sequences (mathematical operations, comparisons), which are commonly used in math-intensive signal processing applications. The DSP units are widely used in a large number of devices, including cell phones, sound cards, modems, hard disks and digital TVs. Single precision floating point arithmetic(IEEE Standard 754) was used in the this first architecture. The main reason that the floating point arithmetic was used, was to keep high the accuracy of the operations needed in the gini gain module calculation, as the final result – the min gini gain computed – may change with a possible loss in precision.

The latency for all the DSPs was the maximum possible as well as their usage in the FPGA board. Table 5-1 summarizes the latencies for the corresponding mathematical operations and the DSPs that each operation occupies. Both Virtex5 boards, LX110T and SX240T that were available to us, do not support the division operation and the conversion from integer to floating point with the DSP units, so the FPGAs logic was used for this reason.

Operation	Latency (cycles)	DSPs occupied
Addition / subtraction	12	2
Multiplication	9	3
Division	28	Logic only
Fixed to float	6	Logic only

Table 5-1: Latency and DSP usage for arithmetic operations

The FPGA's Block RAMs (BRAMs) were used in order to store the BFTree's implementation input that comes from the UDP/IP core and the partial products, sums and frequency counts. The BRAM blocks in structural HDL are configurable memory modules that attach to a variety of BRAM Interface Controllers and their utilization does not require the board's logic blocks, as there are several Block RAMs embedded in each board, where the total size varies from 0.65MB in the LX110T board to 2.26 MB in the SX240T board.

5.2 UDP/IP architecture

In order to connect the hardware implementation which includes both the BFTree and the UDP/IP core implementation with the host PC via the Ethernet cable, a few changes were made to the initial UDP/IP open source code. The overall block diagram of the full implementation is shown in Figure 5-9. For the input/output of the data to/from the FPGA, there are four modes depending on the desirable data width. The offered data width types are 8, 16, 32 and 64 bits for a character, short integer, integer/single precision float number and a double precision float number, respectively. In this architecture the 32bit data width was used so as to fully cover every input case.

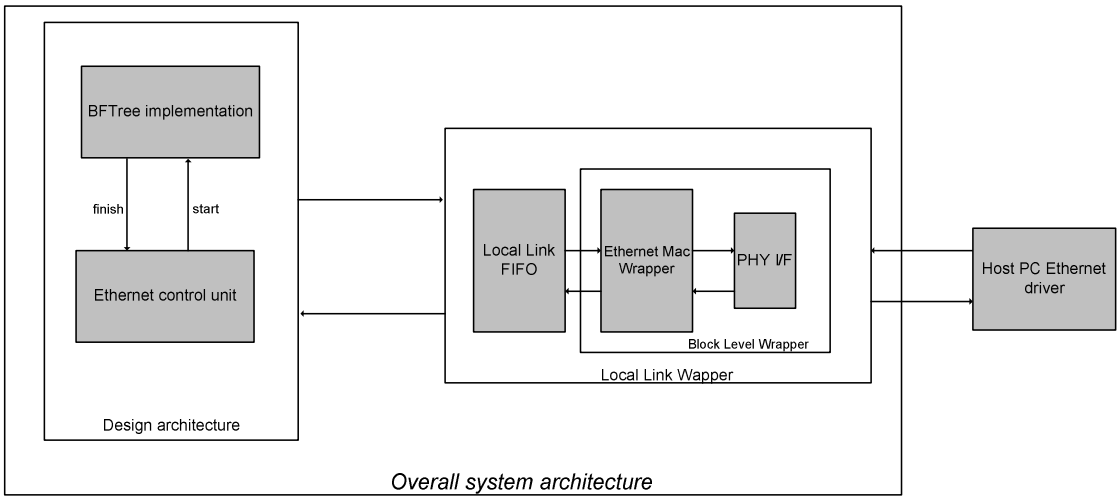

Figure 5-9: Overall system architecture

The basic changes to the initial UDP/IP core implementation were made in the Ethernet control unit. As mentioned above, the BFTree architecture consists of 64 parallel machines each of which is responsible for examining the possible values of input attributes. The Ethernet control unit waits for the Host PC to send a valid data flag in order to receive the 32bit input. In order to 'feed' all the parallel machines with the appropriate input, there are 64 states in the control unit, Figure 5-11, that are responsible to send the received input to the N^{ary} BFTree implementation. At this point, it should be mentioned that in terms of secure synchronization, each one of the 64 states write the input data to FPGA BRAMs and then it is read from the BFTree machine (fig. 5-10). Thus, 64 BRAMs are allocated in the FPGA board for this scope. When all the input data are sent, the host PC sends to the FPGA an end of transmission signal and the processing procedure starts.

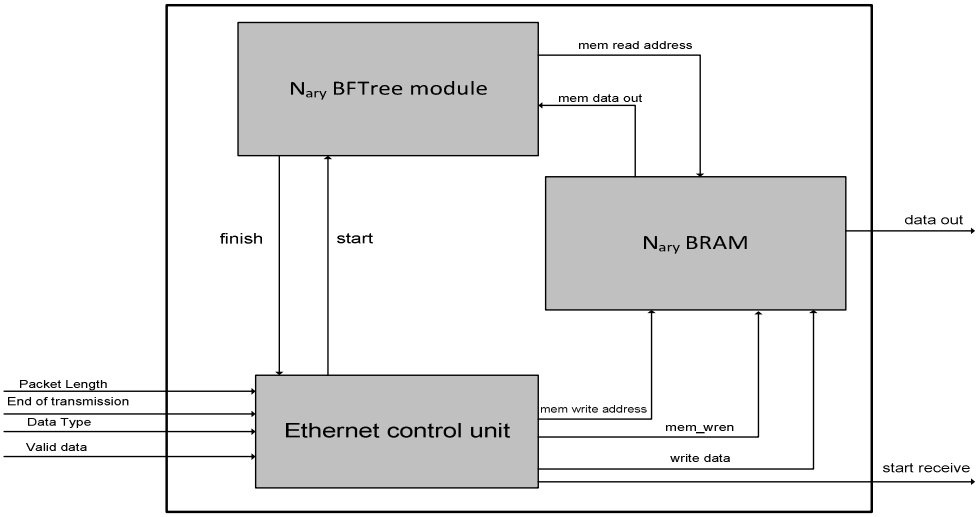

Figure 5-10: Overall system architecture

The processing includes the BRAM read of the corresponding parallel machines and, of course, the previously described procedures for the gini gain calculation. When all the values are examined, the BFTree machine sends to the Ethernet control (fig. 5-11) a finish signal and the transmission process from the FPGA to the PC begins.

The length of the transmission packet to the PC is equal to 1400 bytes, whereas the length of the receive packet from the FPGA is 200 bytes. The Ethernet controller can handle packets up to 1470 bytes, but in order to reassure that the data transmission is lossless, the number is reduced by a few bytes. Moreover, the receive packet length is relatively low, as the only information that the host PC needs is the splitting attribute and the corresponding attribute value combination.

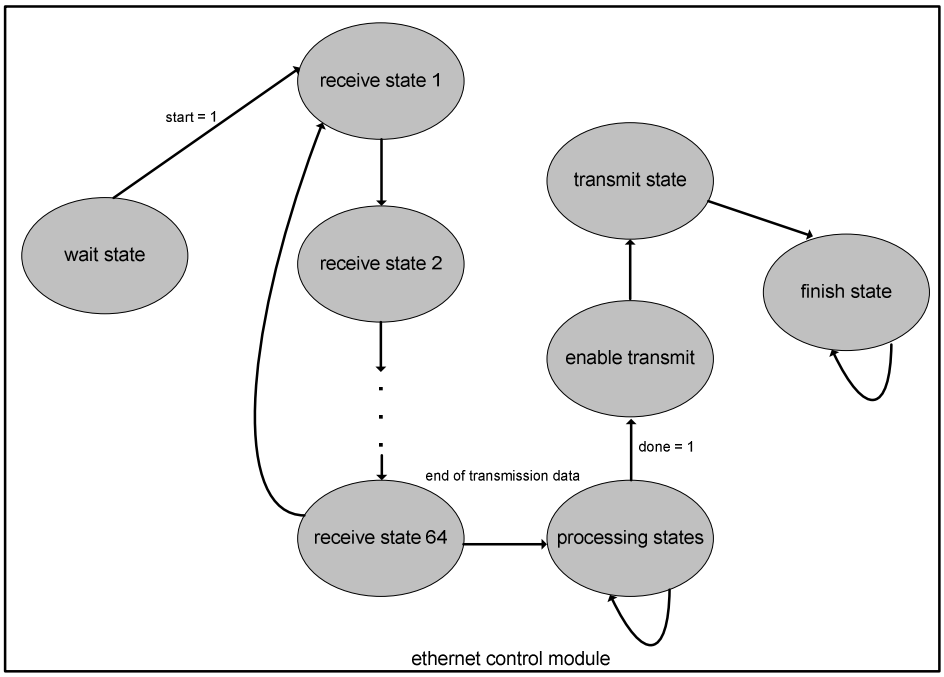

Figure 5-11: Ethernet control module architecture

It should be mentioned that in case of fewer attributes N than the maximum allowed number from the implementation (64 in our case), the Ethernet control module reassures the proper reception of the input data by the FPGA. While the N_{ary} receive state is reached, the control flow continues to the processing states, while it bypasses the remaining 64 – N receive states.

The conversion of the software's implementation input to a binary sequence that can be sent to the BFTree's architecture via the UDP/IP core, was made through a python script. The final input data format is shown in Figure 5-12. The attribute's values as well as the class values are concatenated to a single 32bit number in order to reduce the transmission overhead. To be more specific, as the class and the attribute value can have a maximum of 64 possible values, 6 bits for each representation is adequate. The reason that a 32-bit number is held instead of a 16-bit is that for future expansion of the architecture, 32 bit floating point numbers could be sent, representing an integer/real attribute value. Finally, the Ethernet controller sends the input data to the FPGA until an FFFFFFFF value is met and, therefore, the transmission finishes and the processing of the data by the BFTree machines starts.

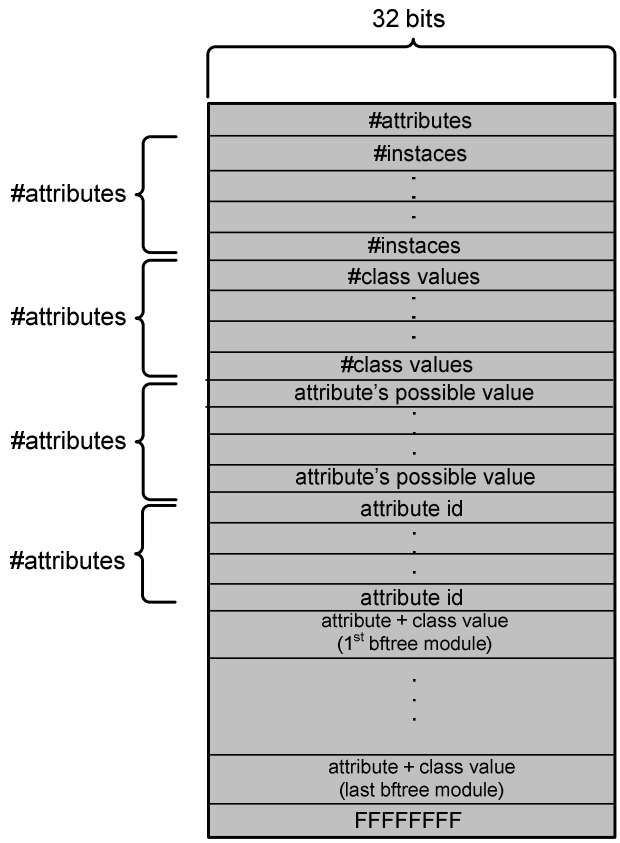

Figure 5-12: Input data format

5.3 Second architecture

This section presents the FPGA-based second architecture of the Best First Tree construction method. The architecture was implemented on two Xilinx Virtex 5 FPGA Family Member Devices, the LX110T and the SX240T. The goal of the implemented system is to compute the best, the maximum in this occasion, Gini gain for all the possible values of the examined attributes in order to find the N^{ary} splitting point and the splitting value of each tree node. The proposed system is implemented in a different architecture than the previously mentioned one, as it exploits some characteristics of the BFTree software implementation in order to have less I/O demands, which adds a significant overhead in the algorithm's execution time. In the following sections, the implementation details will be described.

5.3.1 System's architecture

Figure 5-13 shows the overall proposed architecture of a single BF-Tree construction system, which consists of three main subsystems; Gigabit Ethernet I/O SubSystem, BFTree_8_attributes and Find Max Gini module. The main differentiation made on this architecture vs. the first architecture, is spotted on the BFTree_8_attributes module, whose implementation details will be described below.

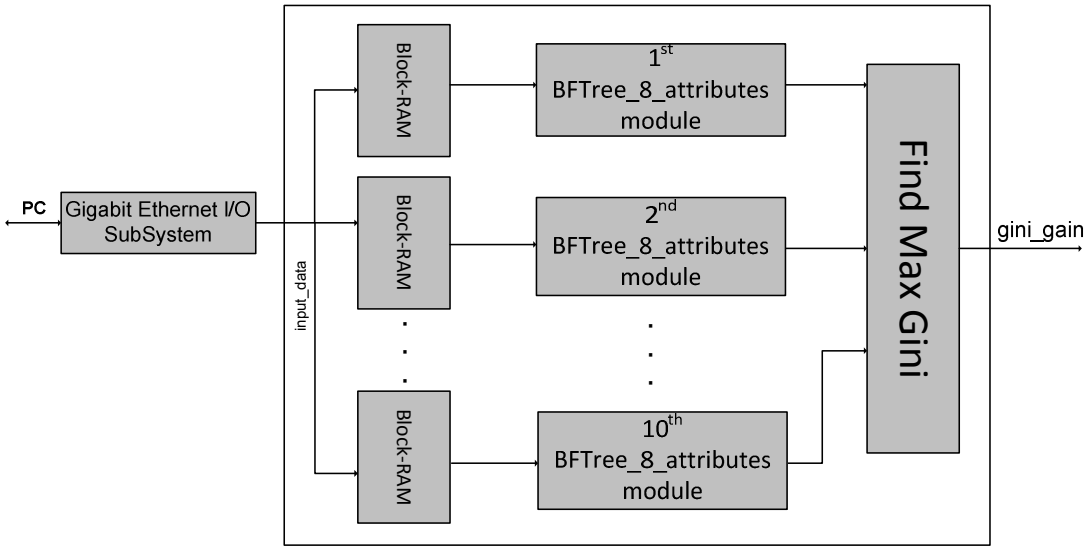

Figure 5-13: The architecture of the FPGA-based BF-Tree construction system

5.3.2 BFTree_8_attributes module architecture

The BFTree_8_attributes module (fig. 5-14) consists of two basic sub-modules; the BFTree_1_attribute and the gini gain calculation module. In order to reduce the resources needed for the basic BFTree module, the gini gain calculation unit was extracted from the basic BFTree machine, as it approximately occupies the 75% of the total resources needed. Thus, each gini gain calculation unit corresponds to eight BFTree modules, allowing us to have a total of 80 parallel machines in the overall architecture. The algorithm's parallelization is accomplished by examining one attribute per BFTree module, achieving a parallelization level of 80 simultaneously examined attributes.

As shown in Figure 5-14, four 8-1 multiplexer is responsible for 'feeding' the gini gain calculation module with the proper corresponding input values. The BFTree_1_attribute module has a throughput greater than 8 cycles for producing the gini gain calculation module inputs, so no synchronization problems will occur and no input values will be lost.

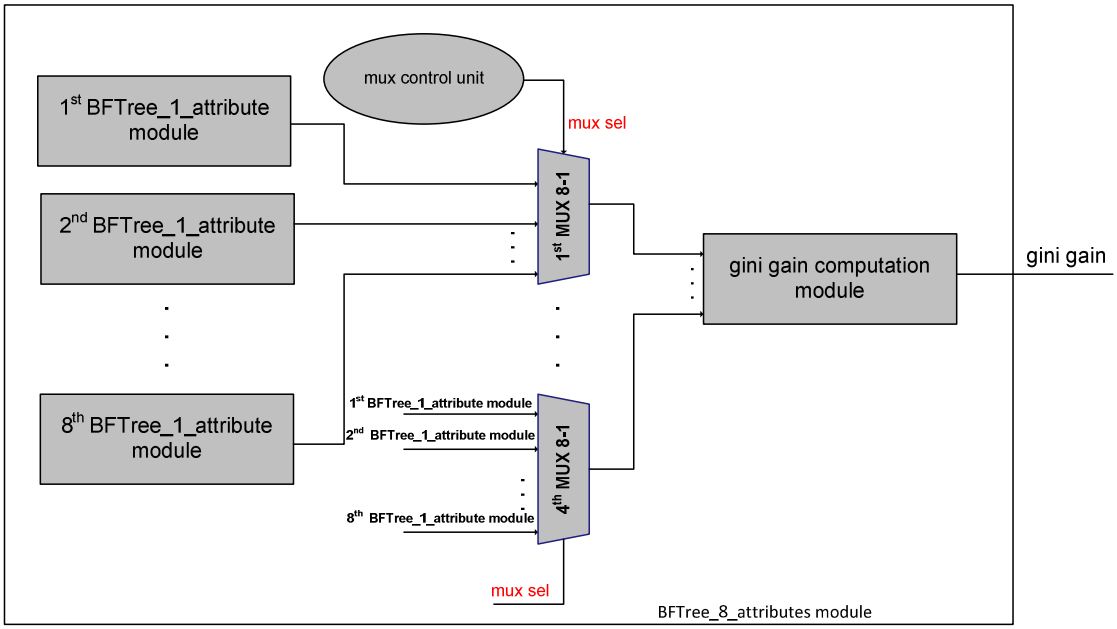

Figure 5-14: The architecture of the BFTree_8_attributes_module module

5.3.3 Gini gain computation module

In the second architecture, the gini gain inputs have changed due to inaccurate results in case of a zero input value. To be more specific, observing the equation (6), it is obvious that with an R_i and L_i class value equal to zero, the calculated gini gain is also equal to zero, which produces divergent results and consequently a different BFTree when compared to the respective software implementation. Thus, the new gini gain that occurred after analyzing the original software's gini equation, (section 4.3.1), is equal to:

$$Gini\ Gain = \frac{\sum_{i=0}^{i<num\_attributes} 64R_i^2}{\sum_{i=0}^{i<num\_attributes} R_i} + \frac{\sum_{i=0}^{i<num\_attributes} 64L_i^2}{\sum_{i=0}^{i<num\_attributes} L_i} =$$

$$= \frac{\sum_{i=0}^{i<num\_attributes} 64R_i^2}{\sum_{i=0}^{i<num\_attributes} R_i} + \frac{\sum_{i=0}^{i<num\_attributes} (64L_i^2 + 64R_i^2) - \sum_{i=0}^{i<num\_attributes} 64R_i^2}{\sum_{i=0}^{i<num\_attributes} (L_i + R_i) - \sum_{i=0}^{i<num\_attributes} R_i} \quad (7)$$

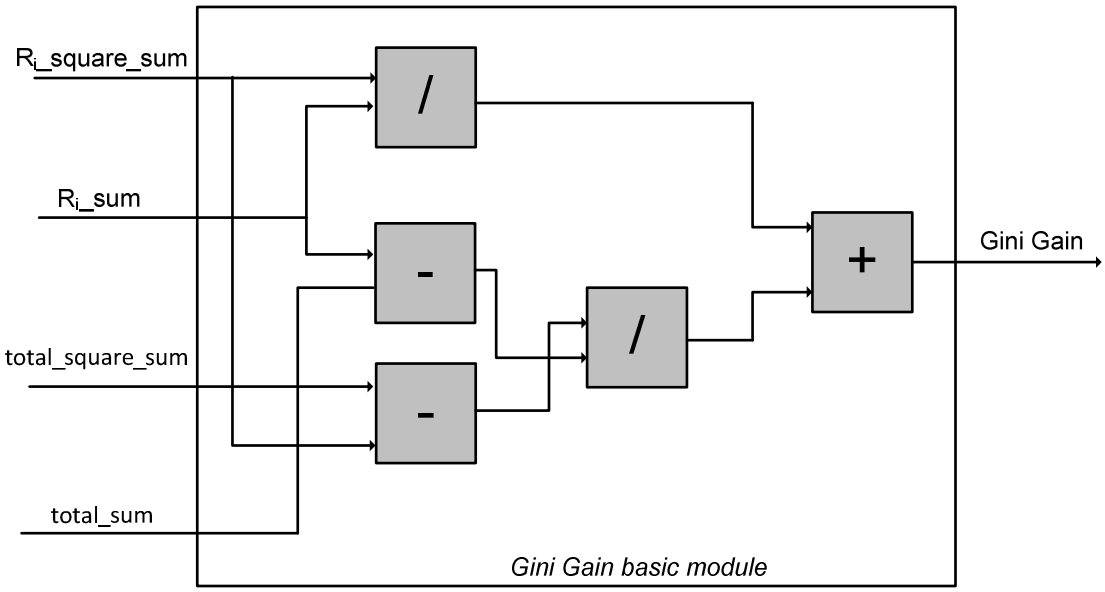

Figure 5-15: Gini gain basic module architecture

It should be mentioned that the software implementation has been altered according to the new gini equation in order to be fully compatible with the hardware implementation and the final results (maximum gini gain, splitting criterion and attribute value combination) were identical to the initial ones.

5.3.4 BFTree_1_attribute module

The BFTree_1_attribute module, Figure 5-15, consists of two main subsystems; control memories and memories_plus_sum_products. The basic inputs and the outputs of this module are the following:

- **Inputs:**
  - Start signal: This signal triggers the BFTree module to start the processing of the input data. It comes from the Ethernet control module and it is activated when the FPGA has received all the required input information from the host PC.

- New_data signal: This signal is activated when a valid 32 bit input was read from the UDP/IP core. It comes from the Ethernet control module and it is used in order to reassure the correct data receiving.

➢ **Outputs:**

- Total_sum signal: This 32 bit signal holds the value of $\sum_{i=0}^{i<num\_attributes} (L_i + R_i)(8)$, whose calculation details will be discussed in the following section.
- Total_square_sum signal: This signal holds the value of $\sum_{i=0}^{i<num\_attributes} (64L_i^2 + 64R_i^2)(9)$
- Square_sum signal: This signal is equal to $\sum_{i=0}^{i<num\_attributes} 64R_i^2 (10)$
- Partial_sum signal: This signal holds the value of $\sum_{i=0}^{i<num\_attributes} R_i (11)$ and combined with the first three outputs, they constitute the gini gain module inputs.
- Done signal: This signal is triggered when the whole algorithm - including the calculation of the last gini gain – finishes its execution. It also informs the Ethernet control unit to start the transmission of the calculated maximum gini gain, the splitting attribute and the attribute's value corresponding combination.

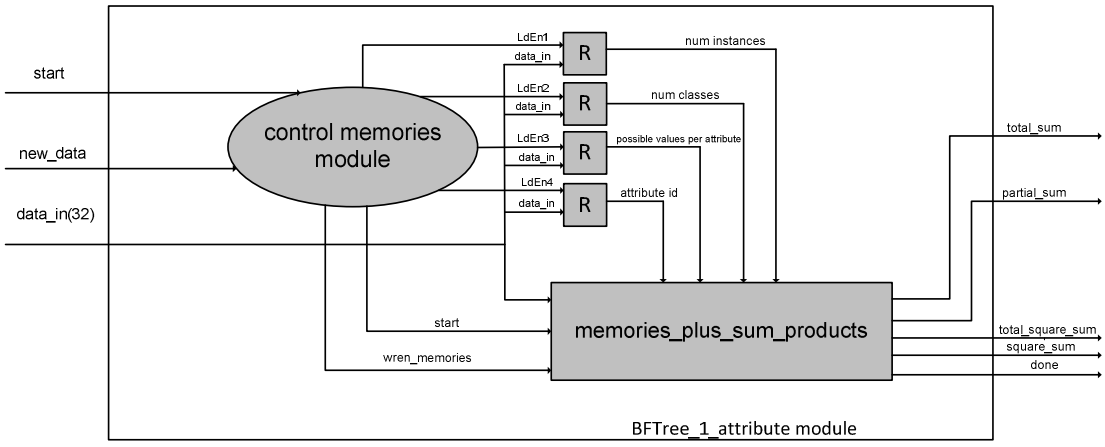

Figure 5-16: BFTree_1_attribute module architecture

5.3.5 Memories_plus_sum_products

The memories_plus_sum_products module (fig. 5-18) consists of six sub modules; mem_4096_32, mem_64_32, square_sum_mem, square_sum_module, add_module and address generator module.

When the start signal from the previously mentioned control unit is equal to 1, the memories_plus_sum_products core starts to count the frequency of each attribute value (mem_64_32) and each attribute value with a specific class value (mem_4096_32). Figure 5-17 shows the structure of the memories and their contents. The input data are the attribute value concatenated with the corresponding class value and they come from the BRAMs that are initialized by the UDP/IP core, as described in section 5.2.

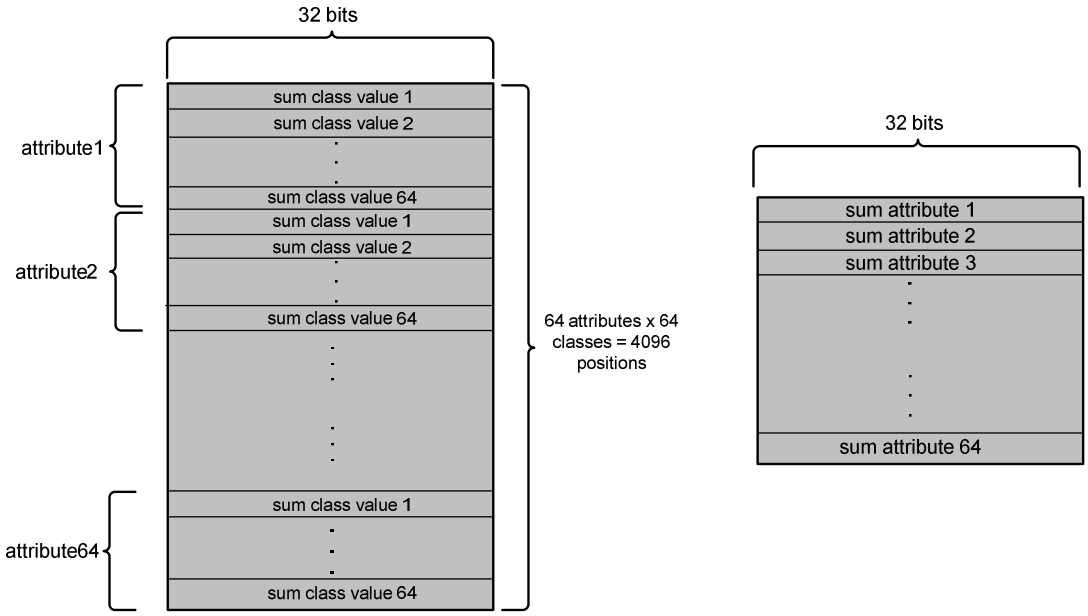

Figure 5-17: Mem_4096_32 and mem_64_32 structure

When the data transmissions has finished and all the data have been written in the two above memories, the input data are read, the square sum module and the add module calculate the total_square_sum_signal, the square_sum_signal (eq. 9, 10), the total_sum_signal and the partial_sum_signal (eq. 8, 11) respectively. Thus, all the required inputs for the gini gain calculation module will be generated according to all the attribute value combinations that come from the address generator module as described in the following section.

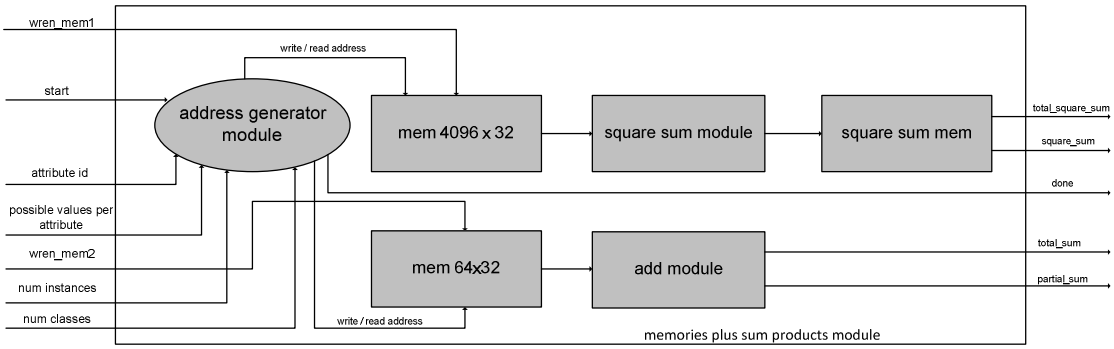

Figure 5-18: Memories plus sum products module structure

5.3.6 Address generator module

The Address generator module (Figure 5-19) is responsible for generating all the possible value combinations of a specific attribute. It takes as input the number of the possible values of each attribute and it outputs the attribute value combination, equal to 64 bits, one for each attribute value and the square sum memory's read address. The way that the described module generates all the possible combinations is the following:

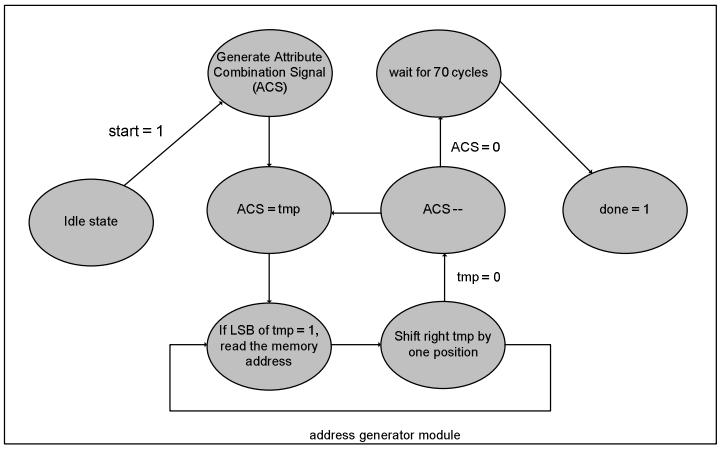

Figure 5-19: Address generator module architecture

A 64bit vector, the attribute_combination_signal is initialized with the N last bits equal to '1', regarding that the examined attribute has N possible values.

Each bit position of the attribute_combination_signal corresponds to the read address of the square sum memory. Thus, with N right shifts to the 64bit signal, all the read addresses for a specific attribute value combination will be acquired. For example if the

attribute_combination_signal holds the 000000….011101 value, then the square sum memory will read the positions 0, 2, 3 and 4 and its corresponding outputs will be multiplied (square_sum_module) in order to produce the final square_sum value, which is the gini gain module's input.

The attribute_combination_signal is decreased by one and the above procedure continues until it reaches the zero value, which means that all the possible gini gain module inputs were generated.

Finally, a few clock cycles after the last gini gain input combinations, the done signal is triggered and the Ethernet controller starts to transmit the BFTree's implementation results to the host PC.

5.4 Comparison of the two architectures

The two architectures described in sections 5.1 and 5.3 implement the basic and the most time consuming function of the BFTree algorithm; the searching of the best splitting criterion and its corresponding splitting value. By finding this splitting criterion, the software can easily and obviously in much less time, build the whole classification tree. There are a few differences between these architectures that will be summarized in this section.

As far as the total size of the input data is concerned, the first architecture is restricted to a few MBs, depending on the FPGA board. This happens due to the fact that all the instances to be examined are initially stored in the BRAMs and then read from the BFTree control. On the other hand, the second architecture has no restriction over the input's size, as when the total number of the instances exceeds the memory's size, the BFTree controller reads the BRAM's data and the same procedure continues until all the instances are read.

Regarding the parallelization level of the algorithm, the first architecture has 64 parallel machines that are processing the different attribute's value combinations of the same attribute. Thus, for each possible combination, the whole input memory should be scanned, adding a significant overhead to the algorithm's running time. The second architecture has 80 parallel machines and each machine calculates the gini gain of a specific attribute by calculating all its possible value combinations. This is feasible due to the information stored in the square sum memory and the alterations made in the gini gain calculation equation. The above two differentiations make the gini gain calculation possible without having to read the input data

more than one time. As a result, for input datasets that have a large number of attributes, the second architecture has less processing time.

The above mentioned differences plus a few more implementation alterations are summarized in table 5-2.

	1st architecture	2nd architecture
Input size	2.3 MB	Unrestricted
#Parallel machines	64	80
Parallelization method	Attribute value combination	Attribute
#Class values	8	64
#Attribute values	64	64

Table 5-2: Summary of the architectures' basic characteristics

6. Systems Performance

This chapter presents the performance of the systems, which are described in the previous chapter, and compares them with the corresponding software implementation in the WEKA platform. Moreover, a few more implementation details are discussed, such as the resource utilization and the validation of the systems in the FPGA device. Finally, a straight comparison between the two reconfigurable logic architectures is made in order to find the most suitable system to be used.

6.1 Resource utilization

This section presents the resources' utilization of both hardware implementations according to the Place and Route report of the Xilinx ISE 10.1 tool. The target board is a Virtex 5 SX240T device and the package is FF1738. Table 6-1 summarizes the resources that are utilized by these architectures. Both architectures include the UDP/IP core, implemented in [58].

Resource	1st architecture (64 parallel machines)	2nd architecture (80 parallel machines)	UDP/IP core
Slice Registers	140,768/149,760 (94%)	138,542/149,760 (93%)	2,151/149,760 (1%)
Slice LUTs	147,253/149,760 (98%)	141,598/149,760 (95%)	1,899/149,760 (1%)
DSPs	1,024/1,056 (97%)	620/1,056 (59%)	0/1,056 (0%)
BRAMs	4/516 (1%)*	72/516 (14%)	21/516 (4%)

Table 6-1: HW architectures' resource utilization (*without taken into consideration the BRAMs utilization)

As shown in Table 6-1, the critical resource for the first architecture is the slice logic utilization. It should be mentioned that the input dataset's size is not taken into consideration in the BRAMs utilization. Thus, a dataset of 2.30 MB is the upper limit for the input's size.

In the second architecture, where 80 parallel machines are mapped in the FPGA device, the critical resource is the logic that is used. As it was explained in the previous chapter, this architecture has no limitation on the input's size. In both architectures, the UDP/IP core resources, Table 6-1 in the last column, are not included.

6.2 Architectures' validation

In both architectures, several test cases were used, shown in Table 6-2, in order to validate the correct operation of the implemented systems. To be more specific, some of the examined inputs were taken by the Weka's official site [52], while the others were generated by the Weka tool generator in order to have a more detailed approach of the systems' characteristics. For the datasets that were generated by the Weka tool, the Bayes distribution was used for the random values that were assigned to each instance. As Table 6-2 presents, the datasets that were used have a wide range of input variables (number of instances, number of attributes, number of possible values /attribute) and thus a safe estimation of the systems' performance can be made.

Datasets	#instances	#attributes	#possible values/ attribute
Nursery	12960	9	4
Mushrooms	8124	23	8
Car	1728	17	2
Bayes_200_4_16	200	4	16
Bayes_200_8_16	200	8	16
Bayes_200_16_16	200	16	16
Bayes_200_32_16	200	32	16
Bayes_200_64_16	200	64	16
Bayes_100_4_4	200	4	4
Bayes_100_4_8	200	4	8
Bayes_100_4_16	200	4	16
Bayes_100_4_32	200	4	32
Bayes_100_32_16	100	32	16
Bayes_200_32_16	200	32	16
Bayes_500_32_16	500	32	16
Bayes_1000_32_16	1000	32	16
Bayes_5000_32_16	5000	32	16
Bayes_10000_32_16	10000	32	16
Bayes_100_4_4	100	4	4
Bayes_250_8_8	250	8	8
Bayes_500_16_16	500	16	16

Table 6-2: Input datasets and their characteristics

In all the datasets used, the splitting values that both systems produced were **identical** to those of the java software implementation and as a result the resulted decision tree was the same as the one produced by the Weka tool.

6.3 Performance of the implemented hardware architectures

In this section, the performance of both previously described architectures is analyzed. Moreover, a straight comparison between the java and the hardware implementation running times is made, as long as a comparison of the two reconfigurable systems' running times.

All the times that are mentioned in the tables are the measured performance on actual designs, completed and downloaded in the respective FPGA platform, and with I/O accounted for. The Weka Java code for the BF-tree construction code was executed on a 4-core Intel Xeon E5430 server with 12 GB RAM and Ubuntu 8.1 OS. The processing system, including the Gigabit Ethernet I/O subsystem is clocked at 102 MHz and 125 MHz for the 1st and the 2nd architecture respectively.

Datasets	SW execution time (sec)	Measured HW time 1st arch. (sec)	HW I/O overhead 1st arch. (sec)	Measured HW time 2nd arch. (sec)	HW I/O overhead 2nd arch. (sec)	Total time 1st arc. (sec)	Total time 2nd arch. (sec)
Nursery	1.050	0.241	0.024	1.532	0.096	0.265	1.628
Mushrooms	1.220	0.872	0.026	2.431	0.104	0.898	2.535
Car	0.140	0.001	0.002	0.592	0.080	0.003	0.672

Table 6-3: Performance of BFTree systems on datasets from [52]

After running the datasets taken from [52], the execution time results showed that for the first architecture a speedup of 1.3 to 46.6 times was achieved, while in the second architecture the running time was increased from 1.5 to 5 times. However, the above datasets are not indicative as the original data mining datasets, as they are not restricted to a size of just a few bytes.

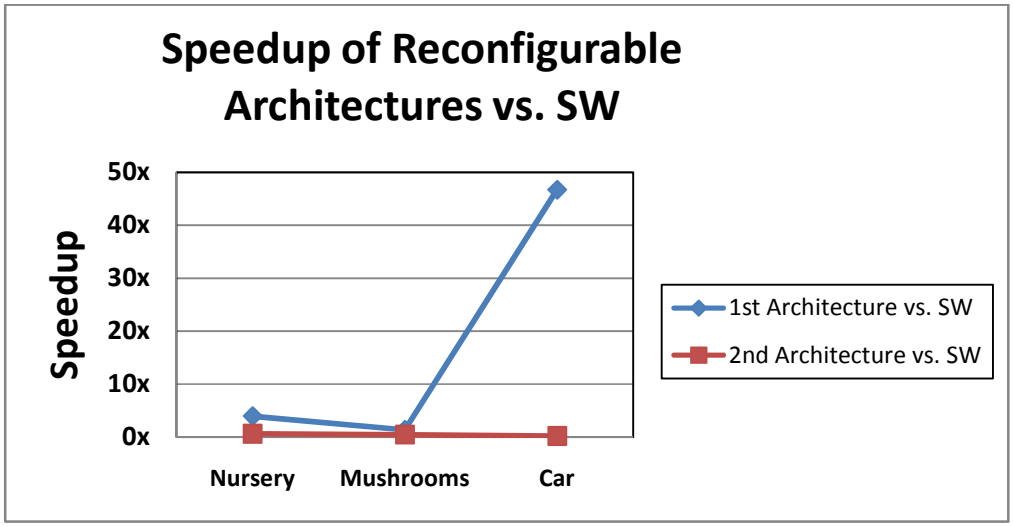

Figure 6-1: Speedup of BFTree systems on datasets from [52]

The software performance is dependent on three basic factors; the number of instances, the number of variables and the number of possible values per each variable of the input dataset. Therefore three different types of real time tests took place and an additional one with three variables (number of attributes, number of possible values, number of instances), showing the performance of the BFTree systems on different experimental cases.

Datasets	SW execution time (sec)	Measured HW time 1st arch. (sec)	HW I/O overhead 1st arch. (sec)	Measured HW time 2nd arch. (sec)	HW I/O overhead 2nd arch. (sec)	Total time 1st arc. (sec)	Total time 2nd arch. (sec)
Bayes_100_32_16	80.20	4.760	0.001	1.478	0.003	4.761	1.481
Bayes_200_32_16	203.00	10.614	0.002	2.891	0.008	10.616	2.899
Bayes_500_32_16	551.54	26.554	0.005	6.683	0.019	26.559	6.702
Bayes_1000_32_16	1574.30	76.305	0.011	12.997	0.047	76.316	13.044
Bayes_5000_32_16	10765.32	358.185	0.071	61.996	0.285	358.256	62.281
Bayes_10000_32_16	25323.58	788.530	0.149	120.668	0.596	788.679	121.246

Table 6-4: Performance of BFTree systems (variable: num of instances)

Table 6-4 shows the **real time measurements** for those test cases that the number of instances increases while the other two "variables" retain stable. The performance speedup that the BFTree systems achieve vs. the software implementation varies from 16.8x to 32.1x for the first architecture and 54.1x to 208.8x for the second architecture, as shown in Figure 6-2.

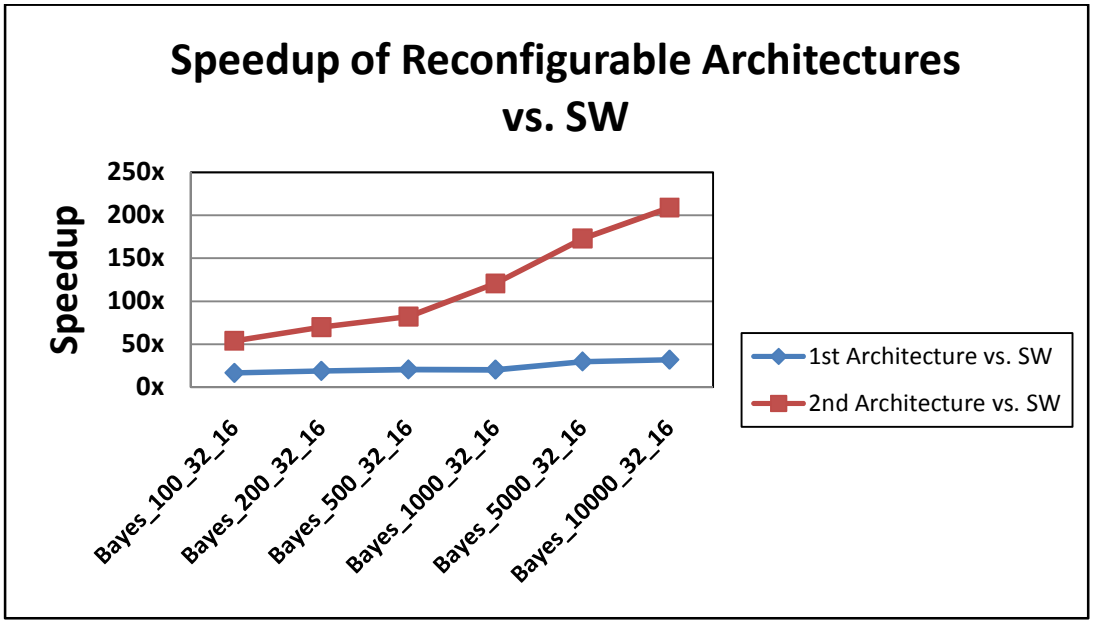

Figure 6-2: Performance of BFTree systems (variable: num of instances)

Datasets	SW execution time (sec)	Measured HW time 1st arch. (sec)	HW I/O overhead 1st arch. (sec)	Measured HW time 2nd arch. (sec)	HW I/O overhead 2nd arch. (sec)	Total time 1st arch. (sec)	Total time 2nd arch. (sec)
Bayes_100_4_4	0.060	$3.0x10^{-5}$	$3.8x10^{-5}$	$1.1x10^{-4}$	$1.5x10^{-4}$	$6.8x10^{-5}$	$2.6x10^{-4}$
Bayes_100_4_8	0.100	$6.1x10^{-4}$	$3.8x10^{-5}$	$5.1x10^{-3}$	$1.5x10^{-4}$	$6.4 x10^{-4}$	$5.2 x10^{-3}$
Bayes_100_4_16	2.641	2.710	$3.8x10^{-5}$	3.424	$1.5x10^{-4}$	2.710	3424.00
Bayes_100_4_32	90729.000	12976	$3.8x10^{-5}$	211106.000	$1.5x10^{-4}$	12976.00	211106.00

Table 6-5: Performance of BFTree systems (variable: num of possible values)

Table 6-5 shows the performance of the implemented system when the number of possible values per variable increases. The reduction of the execution time can be increased by 7 times (1st architecture) for a dataset that its software execution takes more than 25 hours. On the other hand, for the second architecture, the system's running time is increased due to the small number of instances that are used in these datasets.

The reason for the performance decrease in the second architecture is that it is independent of the number of instances that are examined, and thus, for datasets with a few instances the performance drops significantly when compared to the corresponding software implementation.

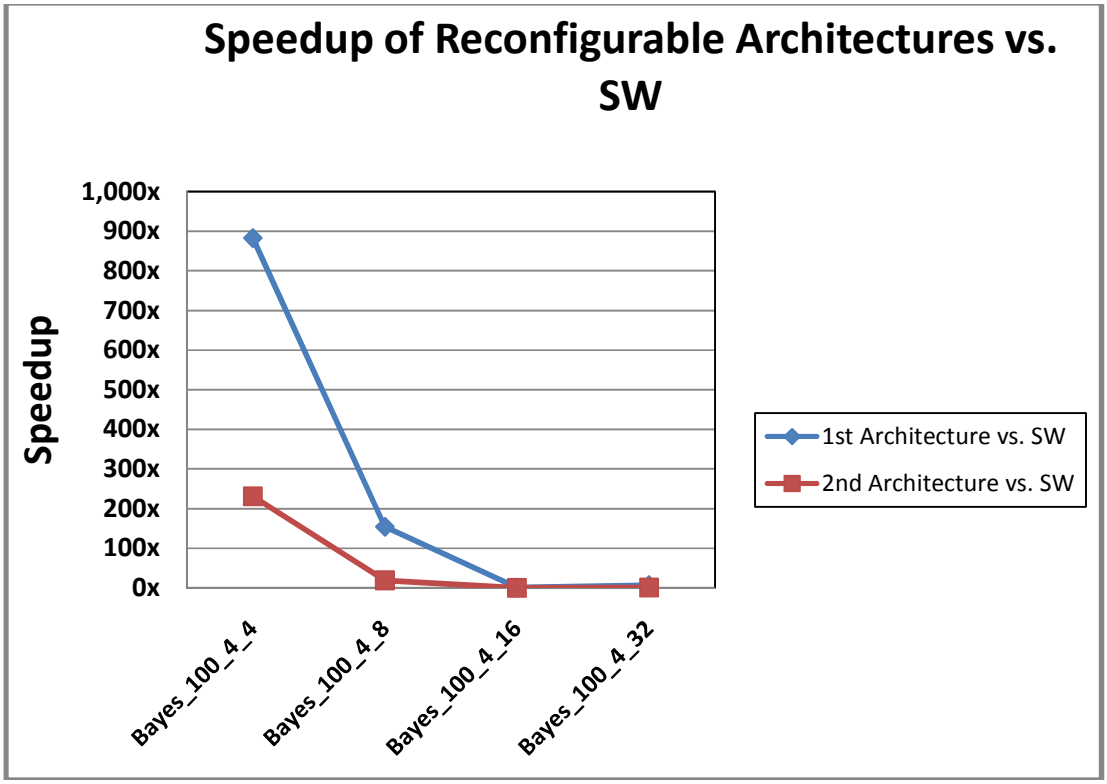

Figure 6-3: Speedup of BFTree systems (variable: num of possible values)

Datasets	SW execution time (sec)	Measured HW time 1st arch. (sec)	HW I/O overhead 1st arch. (sec)	Measured HW time 2nd arch. (sec)	HW I/O overhead 2nd arch. (sec)	Total time 1st arch. (sec)	Total time 2nd arch. (sec)
Bayes_200_4_16	6.050	0.404	$1.1x10^{-4}$	3.241	$4.4x10^{-4}$	0.404	3.241
Bayes_200_8_16	25.890	0.994	$2.8x10^{-4}$	3.245	$1.1x10^{-3}$	0.994	3.246
Bayes_200_16_16	79.630	3.012	$7.5x10^{-4}$	3.252	$3.0x10^{-3}$	3.013	3.255
Bayes_200_32_16	203.00	10.614	$1.9x10^{-3}$	3.252	$7.6x10^{-3}$	10.616	3.260
Bayes_200_64_16	298.00	34.114	$3.5x10^{-3}$	3.260	0.014	34.118	3.274

Table 6-6: Performance of BFTree systems (variable: num of attributes)

Table 6-6 shows the performance of the BFTree systems when the number of attributes increases and the other two "variables" remain stable. The performance speedup of both architectures can reach up to 26 times for the first and 91 times for the second architecture faster than the official software implementation. It should be mentioned that the 2nd architecture offers parallelism in the attributes, as up to 80 attributes can run simultaneously, offering a significant improvement in the system's performance. As a result, the running time of the all the five datasets in table 6-6 remains stable regardless of the attributes' number.

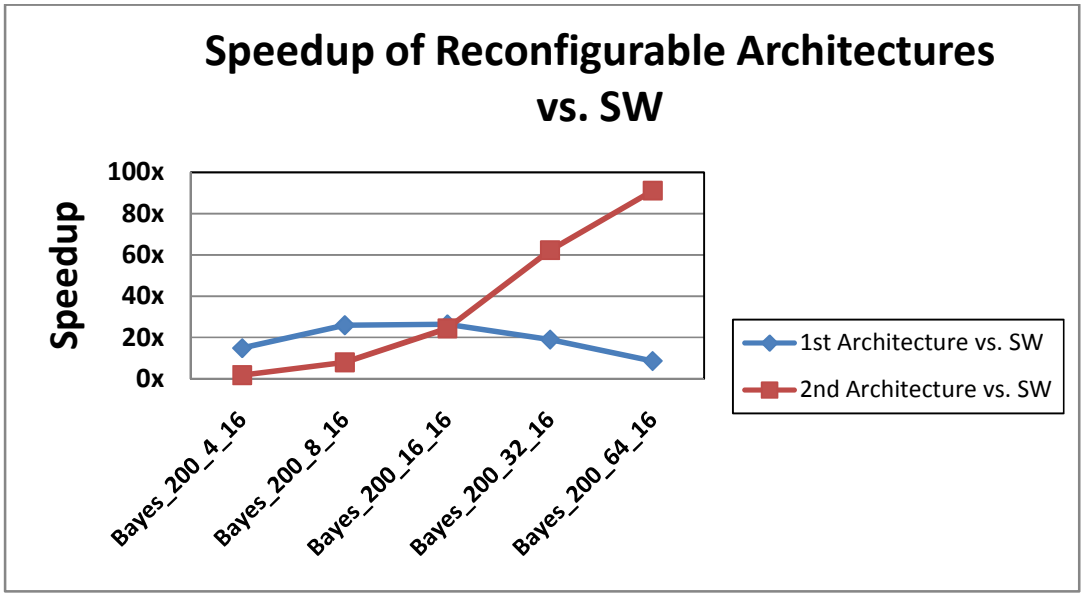

Figure 6-4: Speedup of BFTree systems (variable: num of attributes)

Datasets	SW execution time (sec)	Measured HW time 1st arch. (sec)	HW I/O overhead 1st arch. (sec)	Measured HW time 2nd arch. (sec)	HW I/O overhead 2nd arch. (sec)	Total time 1st arch. (sec)	Total time 2nd arch. (sec)
Bayes_100_4_4	0.060	$3.0x10^{-5}$	$4.0x10^{-5}$	$1.1x10^{-4}$	$1.5x10^{-4}$	$7.0\ x10^{-5}$	$2.6\ x10^{-4}$
Bayes_250_8_8	0.790	0.005	0.001	0.009	0.002	0.006	0.011
Bayes_500_16_16	253.390	7.973	0.003	6.918	0.009	7.976	6.927
Bayes_1000_32_16	1574.300	76.305	0.011	12.997	0.047	76.316	13.044
Bayes_5000_32_16	10765.320	358.185	0.071	61.996	0.285	358.256	62.281
Bayes_10000_32_16	25323.580	788.530	0.149	120.668	0.596	788.679	121.246

Table 6-7: Performance of BFTree systems (variable: num of attributes, num of possible values)

Table 6-7 shows that when all the dataset variables are gradually increased, a speed up from 20-857 times can be achieved in the first architecture, while in the second a speed up from 36 to 230 times. For the dataset with 10000 instances, the second architecture offers a significant running time drop from **7 hours in software to the impressive 2 minutes** in the reconfigurable system.

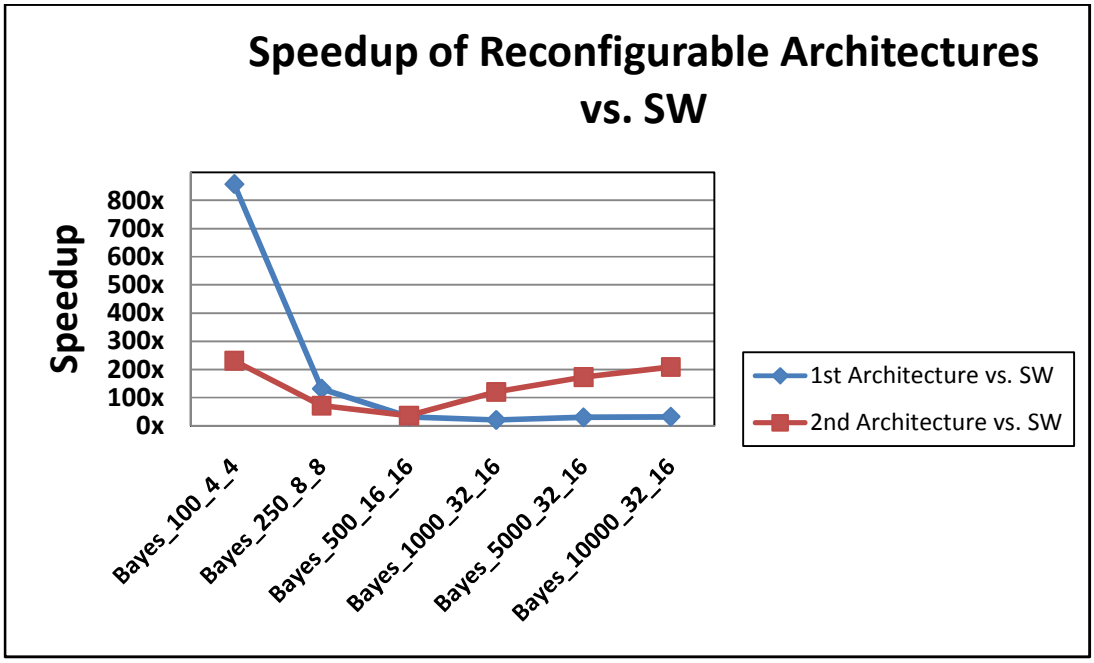

Figure 6-5: Speedup of BFTree systems (variable: num of attributes, num of possible values)

As Tables 6-4 – 6-7 show, the total time for the FPGA-based implemented system, including the I/O overhead, ranges from 6.8×10^{-5} seconds to 3.6 hours for the 1st architecture and 2.6×10^{-4} seconds to 59 hours for the 2nd architecture, while the software execution time ranges from 0.06 seconds to 25 hours. The main factor that affects the running time of the systems is the increase of the number of input instances being examined and the corresponding number of attributes of each instance. Furthermore, the increase of the possible values of each attribute significantly affects the software running time, as the number of iterations executed in order to calculate all the possible values is exponentially augmented.

6.4 Quality performance evaluation of the two implemented systems

In this section, a final evaluation of the two implemented systems will be made, as well as an examination of the test case scenarios that best suit to each system.

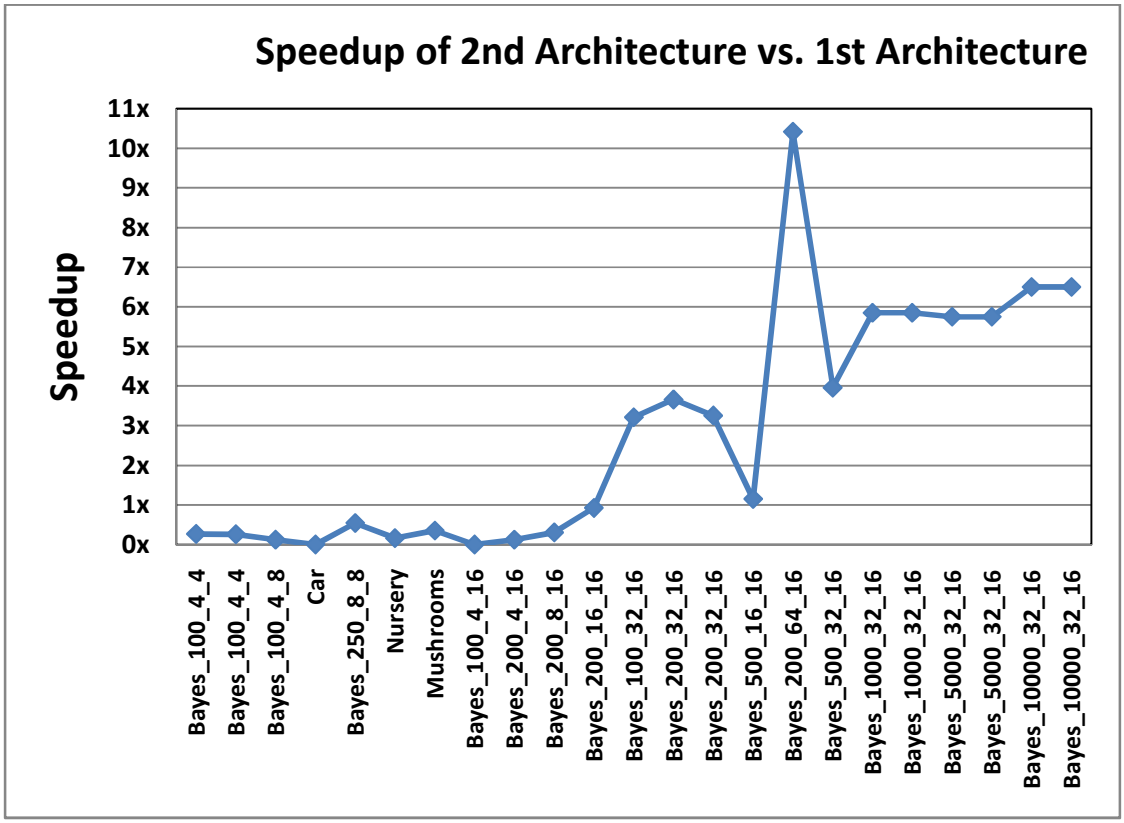

Figure 6-6: Speedup of the 2nd BFTree system when compared with the 1st architecture (the datasets are placed in ascending order as far as their SW execution time)

The performance speedup that the BFTree system can achieve vs. the official software implementation varies from one order of magnitude up to almost three orders of magnitude for the real time measurements on different nature datasets. It is important to mention that the above Tables show that the implemented systems can dramatically reduce the execution time of building the decision tree model from one day to 3 hours. Finally, as figure 6-6 shows, the first architecture is proposed for datasets with a few instances (up to 100) or for datasets with possible values per attribute up to four.

On the other hand, for datasets with a large number of instances, combined with possible values per attribute more than four, the second architecture can achieve significantly better running times when compared to both the software implementation and even to the first architecture. Thus, taking into consideration the above observations, a **dynamically reconfigurable system** which utilizes *the first architecture when the number of instances or the possible values per attribute is minimal* and the *second architecture for cases with huge dataset, as far as the*

number of instances, or high dimensional datasets, will offer an optimal performance in the construction of the decision tree.

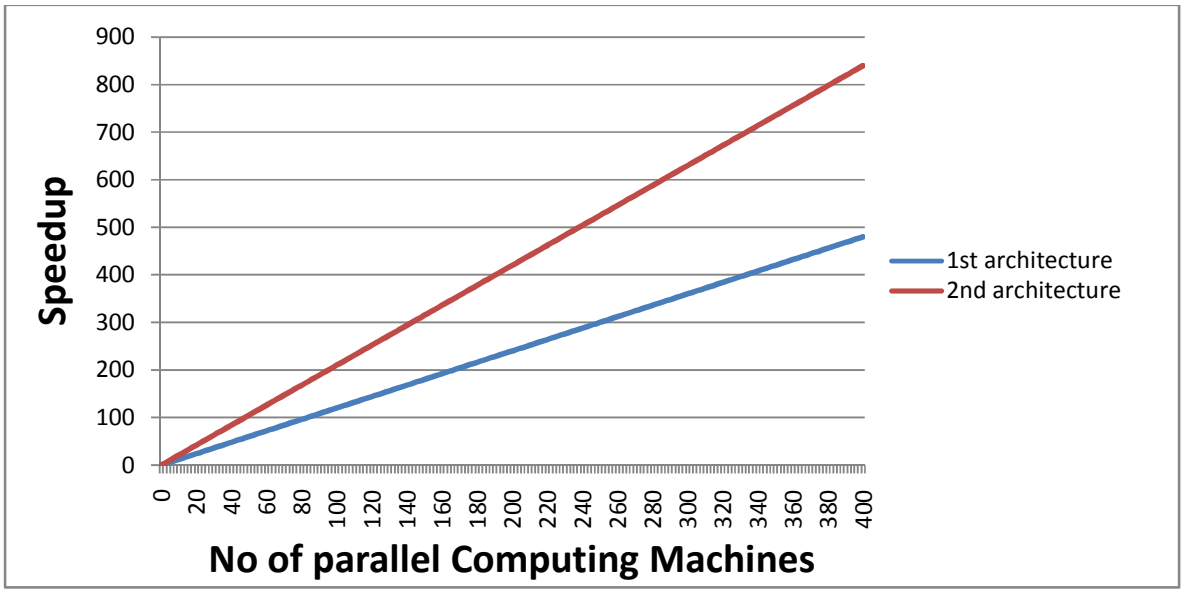

Figure 6- 7: Quality performance evaluation of the two architectures (variable: num of parallel machines)

As figure 6-7 shows, the performance of both architectures **increases linearly** according to the number of parallel machines. Thus, suppose having a board that can map 400 parallel machines, the second architecture can offer double speedup when compared to the first architecture, while simultaneously it can offer a speedup of approximately 9 times when compared to the already implemented architecture that maps 80 parallel machines.

In the previous section, the performance of the two implemented systems was examined by altering one of the three input variables – number of instances, number of attributes and possible values per attribute – while keeping stable the others. Regarding the number of attributes and the number of possible values, all the possible scenarios were taken into consideration as long as the systems can handle up to 64 possible values and 64 attributes. On the other hand, the systems' behavior while increasing the number of instances was not examined, as the datasets that ran on the FPGA device were up to 10,000 instances. Figure 6-8 makes a projection of the performance of both systems regarding the number of instances, based on the real time measurements that were made from the above mentioned datasets.

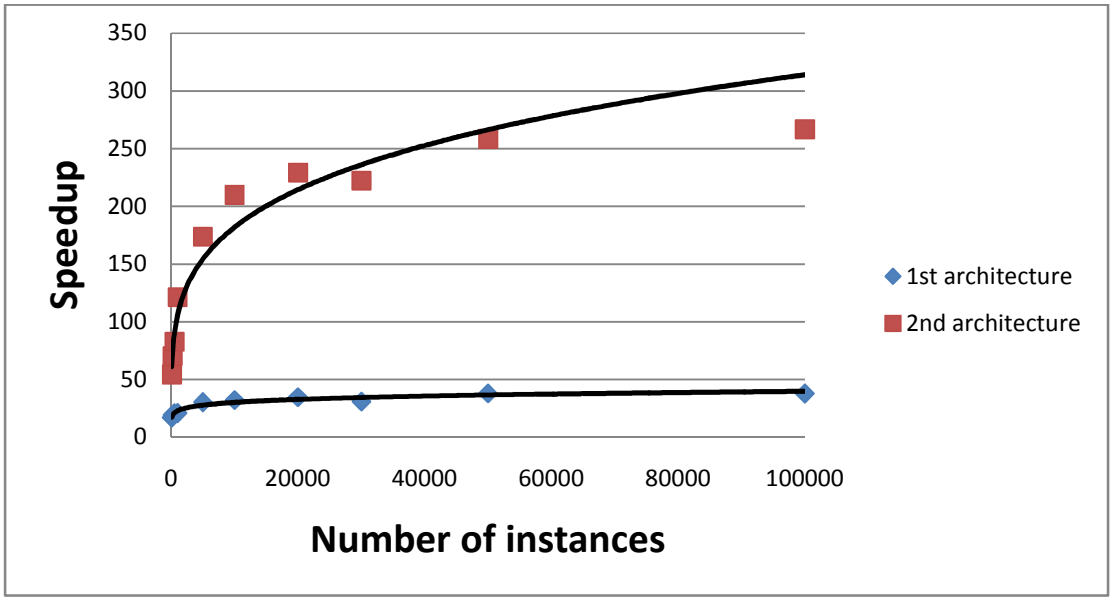

Figure 6- 8: Quality performance evaluation of the two architectures (variable: num of instances)

As figure 6-8 shows, for very large datasets, the second architecture offers a significant speedup increase of the measured running time that can reach up to 8 times greater than the corresponding speedup offered by the first architecture.

7. Conclusions and Future work

Concluding, this Master of Science thesis worked on accelerating the construction of DTC models using reconfigurable logic. It should be mentioned that decision trees can be used in many different disciplines including medical diagnosis, cognitive science, artificial intelligence, game theory, engineering, and of course data mining. For example, in medical decision making (classification, diagnosing, etc.) there are many situations where decisions must be made effectively and reliably. Conceptual simple decision making models with the possibility of automatic learning are the most appropriate for performing such tasks. Generally, decision learning is ubiquitous in:

- *medical diagnosis*: identify new disorders from observations

- *loan applications*: predict risk of default

- *prediction: (*climate, stocks, etc.) predict future from current and past data

- *speech/object recognition*: from examples, generalize to others

Decision trees are a reliable and effective decision making technique that provide high classification accuracy with a simple representation of gathered knowledge and are used in different areas of decision making. Thus, this work can be expanded into the above mentioned fields and therefore effectively accelerate various decision tree construction based applications.

This Master of Science thesis presents two FPGA-based implementations of the widely-used Best First Decision Tree (BF-Tree) classification method. As described above, the implementations offer a performance speedup that can reach up to three orders of magnitude when compared to WEKA's Java code execution time on a state-of-the-art multi-core CPU. Moreover, the implemented system is at least two to four times faster than previous works that solve the same problem for input datasets that demand huge execution times. It is important to mention that the proposed architectures can offer much higher performance vs. official software for higher dimension input datasets that are mainly used in common data mining applications.

This project is almost a complete work whereas there are some points that can be extended in order to achieve better results and performance. Below there are some ideas that can be proposed to extend the subject of this work; first, the replacement of the UDP/IP core with a core (i.e. PCI-Express) that offers higher transmission I/O rates to and from the FPGA would offer an improvement in the system's performance.

Moreover, in this work the inputs with nominal values are processed by the implemented system, while the numeric values from the existing java software implementation. An implementation extension to handle the numeric values in reconfigurable logic would offer a more integrated and less time consuming system.

The main goal of this thesis was to study and propose a more generic decision tree construction method and not to simply implement a decision tree. Thus, as a result, a few capabilities that the DTC method offers were not taken into consideration. Consequently, a pre/post pruning processing of the decision tree or a more efficient management of the missing values that may occur in the input dataset, would have also offered a more complete system. Also, the implementation of the whole tree in the reconfigurable logic, without having the interference of the software, would have resulted in a more efficient embedded system. Finally, after studying and analyzing both implemented architectures, a capability of dynamically reconfiguring the FPGA device to swap between the two architectures depending on the input datasets, would have resulted in much less running time from the hardware, thus totally outperforming the existing software implementation.

8. References

[1] G. Piatetsky-Shapiro, W. J. Frawley, “Knowledge Discovery in Databases,” AAAI/MIT Press, Cambridge, 1991.

[2] R. Agrawal, T. Imieli, A. Swami, “Mining association rules between sets of items in large databases,” ACM SIGMOD international conference on Management of data, 207-216, 1993.

[3] M. Leese, S. Landau, “Model-based clustering using S-PLUS,” International Journal of Methods in Psychiatric Research, Vol. 15, 146–156, 2006.

[4] http://www.spss.ch/upload/1122644952_The%20SPSS%20TwoStep%20Cluster%20Component.pdf

[5] http://www.sas.com/industry/retail/intelligent-clustering/index.html#section=1

[6] J. Hartigan, M. Wong, "Algorithm AS 136: A K-Means Clustering Algorithm," Journal of the Royal Statistical Society, Series C (Applied Statistics) Vol. 28, 100–108, 1979.

[7] H.-S.Park, C.-H. Jun, “A simple and fast algorithm for K-medoids clustering,” Expert Systems with Applications, Volume 36, Issue 2, Part 2, 3336-3341,2009.

[8] G. Karypis, E.-H. Han, V. Kumar, "Chameleon: hierarchical clustering using dynamic modeling," Computer, vol.32, no.8, 68-75, 1999.

[9] T. Zhang, R. Ramakrishnan, M. Livny, “BIRCH: an efficient data clustering method for very large databases,” SIGMOD Rec. 25, 103-114, 1996.

[10] D. Birant, A. Kut, “ST-DBSCAN: An algorithm for clustering spatial–temporal data,” Knowledge Engineering, Volume 60, Issue 1, 208-221,2007.

[11] M. Ankerst, M. Breunig, H.-P. Kriegel, J. Sander, “OPTICS: ordering points to identify the clustering structure,” Proceedings of the 1999 ACM SIGMOD international conference on Management of data (SIGMOD), 49-60, 1999.

[12] A. Hinneburg, H. Gabriel, “Denclue2.0: Fast Clustering Based on Kernel Density Estimation,” International symposium on intelligent data analysis, 70-80, 2007.

[13] W. Wang, J. Yang, R. Muntz, “STING: A Statistical Information Grid Approach to Spatial Data Mining”, 23th International Conference on Very Large Data Bases, Morgan Kaufmann Publishers, 186-195, 1997.

[14] G. Sheikholeslami, S. Chatterjee, A. Zhang, "'WaveCluster: A Multi-Resolution Clustering Approach for Very Large Spatial Databases", 24th International Conference on Very Large Data Bases, 428 – 439, 1998.

[15] N. Mustapha, M. Jalali, M. Jalali, "Expectation Maximization Clustering Algorithm for User

Modeling in Web Usage Mining Systems," European Journal of Scientific Research, Vol. 32, No. 4, 467-476, 2009.

[16] D. H. Fisher, "Improving inference through conceptual clustering," Proceedings of the AAAI Conferences, 461–465, 1987.

[17] J. Vesanto, E. Alhoniemi, "Clustering of the Self-Organizing Map," IEEE TRANSACTIONS ON NEURAL NETWORKS, VOL. 11, NO. 3, 586-600, 2000.

[18] K. Santhisree, A. Damodaram, "CLIQUE: Clustering based on density on web usage data: Experiments and test results,"3rd International Conference on Electronics Computer Technology (ICECT), vol.4, 233-236, 2011.

[19] C. Aggarwal, J. Wolf, P. Yu, C. Procopiuc, J. Park, "Fast algorithms for projected clustering," ACM SIGMOD international conference on Management of data (SIGMOD), 61-72, 1999.

[20] N. Slonim, N. Friedman, N. Tishby, "Agglomerative Multivariate Information Bottleneck," Neural Information Processing Systems Conference (NIPS), 617-623, 2000.

[21] R. Agrawal, R. Srikant, "Fast algorithms for mining association rules in large databases," 20th International Conference on Very Large Data Bases, VLDB, 487-499, Santiago, 1994.

[22] Jiawei Han, Jian Pei, Yiwen Yin, and Runying Mao. 2004. Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach. Data Min. Knowl. Discov. 8, 1 (January 2004), 53-87.

[23] M. Zaki, "Scalable algorithms for association mining," IEEE Transactions on Knowledge and Data Engineering, 12(3), 372–390, 2000.

[24] R. Quinlan, "Induction of decision trees," Machine Learning, Vol. 1, 81–106, 1986.

[25] E. Hunt, J. Marin, ET. Stone, "Experiments in induction," New York: Academic Press, 1966.

[26] J. R. Quinlan, "C4.5: Programs for Machine Learning," Morgan Kaufmann Publishers, 1993

[27] L. Breiman, J. Friedman, C. Stone, R. Olshen, "Classification and Regression Trees," Chapman & Hall, 1984.

[28] S. Russell, P. Norvig, “Artificial Intelligence: A Modern Approach,” Prentice Hall, 2006.

[29] B. Boser, I. Guyon, V. Vapnik, “A training algorithm for optimal margin classifiers,” 5th annual workshop on Computational learning theory (COLT), ACM, 144-152, 1992.

[30] V. Vapnik, “The Nature of Statistical Learning Theory (Information Science and Statistics), Springer Verlag, 2000.

[31] B. Liu, W. Hsu, Y. Ma, “Integrating Classification and Association Rule Mining,” KDD, 1998.

[32] W. Li, J. Han, J. Pei, "CMAR: accurate and efficient classification based on multiple class-association rules," Proceedings IEEE International Conference on Data Mining, 369-376, 2001.

[33] X. Yin, J. Han, "CPAR: Classification based on Predictive Association Rules,” SDM, 2003.

[34] S. Sun, M. Steffen, J. Zambreno, “A Reconfigurable Platform for Frequent Pattern Mining,” International Conference on Reconfigurable Computing and FPGAs (RECONFIG), IEEE Computer Society, 55-60, 2008.

[35] S. Sun, J. Zambreno, “Mining association rules with systolic trees,” International Conference on Field Programmable Logic and Applications (FPL), pages 143–148, 2008.

[36] Z. Baker, V. Prasanna, “An Architecture for Efficient Hardware Data Mining using Reconfigurable Computing Systems,” 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), IEEE Computer Society, 67-75, 2006.

[37] Z. Baker, V. Prasanna, “Efficient Hardware Data Mining with the Apriori Algorithm on FPGAs,” 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), IEEE Computer Society, 3-12, 2005.

[38] T. Saegusa, T. Maruyama, "An FPGA implementation of real-time K-means clustering for color images", Real-Time Image Processing Journal, 309-318, 2007.

[39] M. Estlick, M. Leeser, J. Theiler, J. Szymanski, “Algorithmic transformations in the implementation of K-means clustering on reconfigurable hardware,” ACM/SIGDA 9th international symposium on Field programmable gate arrays (FPGA), ACM, 103-110, 2001.

[40] R. Narayanan, D. Honbo, G. Memik, A. Choudhary, J. Zambreno, “Interactive presentation: An FPGA implementation of decision tree classification,” Conference on Design, automation and test in Europe (DATE), EDA Consortium, 189-194, 2007.

[41] J. Shafer, R. Agrawal, M. Mehta, “SPRINT: A scalable parallel classifier for data mining,” International Conference on Very Large Databases (VLDB), 1996.

[42] C. Wolinski, M. Gokhale, K. McCabe, “A reconfigurable computing fabric,” Engineering of Recongurable Systems and Algorithms Conference (ERSA), 2004.

[43] R. Struharik, L. Novak, “Intellectual property core implementation of decision trees,” IET Comput. Digit. Tech. 3, 259, 2009.

[44] A. Bermak, D. Martinez, “A compact 3D VLSI classifier using bagging threshold network ensembles,” IEEE Trans. Neural Networks, Vol. 14, 1097–1109, 2003.

[45] S. Lopez-Estrada, R. Cumplido, “Decision tree based FPGA architecture for texture sea state classification,” Reconfigurable Computing and FPGA’s (ReConFig), IEEE International Conference, 1–7, 2006.

[46] L. Rokach, O. Maimon, “Top-down induction of decision trees – a survey”, IEEE Trans. Syst. Man Cybern., 476–487, 2005

[47] W. Ma and G. Agrawal, “A translation system for enabling data mining applications on GPUs,” 23rd international conference on Supercomputing, 2009.

[48] W. Fang, M. Lu, X. Xiao, B. He, Q. Luo, “Frequent itemset mining on graphics processors,” 5th International Workshop on Data Management on New Hardware (DaMoN), ACM, 34-42, 2009.

[49] M. Papadonikolakis, C. Bouganis, “A scalable FPGA architecture for non-linear SVM training,” International Conference on Field-Programmable Technology, 337-340, 2008.

[50] V. Vapnik, “The Nature of Statistical Learning Theory,” Springer Verlag, 1995.

[51] M. Papadonikolakis, C. Bouganis, G. Constantinides, “Performance comparison of GPU and FPGA architectures for the SVM training problem,” International Conference on Field-Programmable Technology, 388-391, 2009.

[52] http://www.cs.waikato.ac.nz/ml/weka/

[53] C. Brudson, “Exploratory spatial data analysis and local indicators of spatial association with XLISP-STAT,” Blackwell Publishers Ltd, 1998

[54] http://www.kdnuggets.com/datasets/

[55] T.J. Monk, S. Mitchell, L.A. Smith, G. Holmes, “Geometric comparison of classifications and rule sets,” AAAI workshop on knowledge discovery in databases, 1994.

[56] D. Coppersmith, S. Hong, J. Hosking, “Partitioning Nominal Attributes in Decision Trees,” Data Mining and Knowledge Discovery, an International Journal, Kluwer Academic Publishers, Vol.3, 197-217, 1999.

[57] I. Witten, E. Frank, “Data Mining: Practical Machine Learning Tools and Techniques,” Morgan Kaufmann 2nd edition, 2005.

[58] N. Alachiotis, S.A. Berger, A. Stamatakis, “Efficient PC-FPGA Communication Over Gigabit Ethernet,” IEEE Computer and Information Technology, 1727 – 1734, 2010.