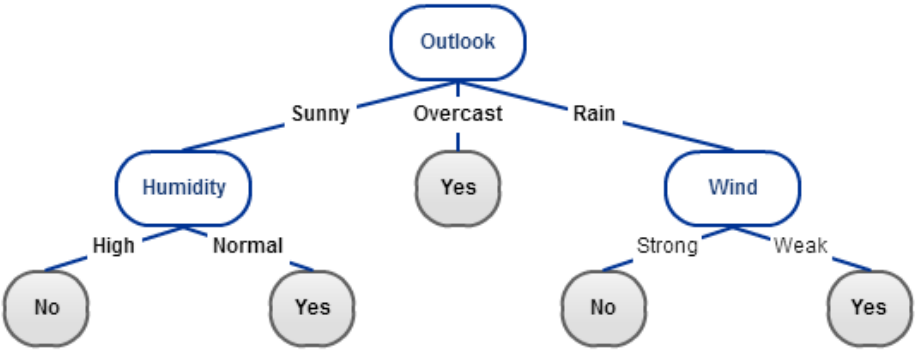
TÉCNICO LISBOA



Classification On The Cloud Using MapReduce

Simão Miguel Anjo Martins

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Doctor Cláudia Martins Antunes

Examination Committee

Chairperson: Doctor José Carlos Alves Pereira Monteiro

Supervisor: Doctor Cláudia Martins Antunes

Members of the Committee: Doctor Alexandre Paulo Lourenço Francisco

November 2014

RESUMO

Na última década empresas acumularam grandes quantidades de dados, e têm vindo a tirar proveito dessas bases de dados usando algoritmos de *data mining* e *machine learning* (DM-ML). No entanto devido ao recente crescimento exponencial do tamanho das bases de dados computadores individuais já não conseguem ligar com dados com esta dimensão, por isso novas soluções são precisas.

Computação paralela e distribuída parece oferecer a solução, no entanto adaptar algoritmos de DM-ML para que estes funcionem num ambiente distribuído e paralelo não é uma tarefa trivial.

Recentemente um novo modelo de programação chamado *MapReduce* foi proposto. Este modelo permite a implementação de algoritmos e tarefas de processamento num ambiente distribuído e paralelo de forma fácil.

Com este trabalho vamos mostrar como implementar um classificador, chamado MRID4, usando o modelo de programação *MapReduce*. Este classificador é fortemente baseado no algoritmo ID3 mas também é capaz de lidar com atributos contínuos da mesma forma que o algoritmo C4.5.

Esta implementação não é trivial, visto que a abstracção que o *MapReduce* providencia não é completamente transparente. Isto obriga o programador a ter que se preocupar com aspectos provenientes de um ambiente distribuído, aspectos esses que vão desde como implementar um algoritmo de divisão e conquista que funciona de forma eficiente sobre o *MapReduce*, até, como garantir que os dados que são computados na função *map* são, de alguma forma, agregáveis no *reduce* e que o resultado final corresponde ao mesmo que se o algoritmo fosse corrido de forma sequencial num só computador.

Os resultados experimentais que obtemos mostram que o nosso algoritmo tem uma grande escalabilidade.

PALAVRAS CHAVE

Data mining, MapReduce, classificação, Hadoop, ID3

ABSTRACT

In the last decade companies have accumulated vast amounts of data, and have been reaping the benefits of such databases by using data mining and machine learning (DM-ML) algorithms. However due to recent exponential increase in databases sizes, single computers can no longer deal with these enormous datasets and new solutions are required.

Parallel and distributed computing seems to offer the solution; however, adapting DM-ML algorithms to work in a parallel and distributed environment is no trivial task. So, a new programming model called MapReduce has been proposed. This model allows for an easy implementation of algorithms and processing tasks in a parallel and distributed environment.

With this work will show how to implement a classifier, the MRID4, using the MapReduce programming model. This classifier is heavily based on the ID3 algorithm but it also ties together the idea of C4.5 on how to handle continuous attributes.

Such implementation is by no means trivial, as the abstraction provided by MapReduce is still leaky. This leads to the programmer having to be concerned about some of the aspects that come in a distributed environment. These aspects range from how to devise a divide and conquer implementation of the algorithm that works efficiently on MapReduce, to, how to make sure that the computed data in the map is somehow aggregable in the reduce and that is corresponds to the same output as when the algorithm is ran in a non-distributed environment.

The obtained experimental results show that our algorithm has a very good scalability.

KEYWORDS

Data mining, MapReduce, Hadoop, ID3, classification.

ACKNOWLEDGEMENTS

I would like to thank everyone who have helped me through the writing of this dissertation and that in some way have contributed to make it possible.

To my supervisor, Doctor Cláudia Antunes, for all the guidance, patience and knowledge she provided.

To my father and brothers, for all the proofreading, spell checking, encouragement and brainstorming.

To my friends, who have helped me during hard phases of the work, and that always showed to be available to help.

To all the folks at Sky.fm Smooth Jazz and ETN.fm that helped me through long stretches of time, by keeping me entertained with some great music.

And finally to the educare project (PTDC/EIA-EIA/110058/2009) which provided me with a research grant.

Thank you all.

CONTENTS

FIGURES INDEX

TABLES INDEX

EQUATIONS INDEX

ALGORITHMS INDEX

Chapter I INTRODUCTION

In the last decade companies have accumulated vast amounts of data. But only very recently have they discovered the potential that lies within it. This potential has been harvested through data mining and machine learning (DM-ML) algorithms, however due to recent exponential increase in databases sizes, single computers can no longer deal with these enormous datasets and new solutions are required. Parallel and distributed computing seems to offer the solution; however adapting DM-ML algorithms to work in a parallel and distributed environment is no trivial task.

In recent years a new programming model called MapReduce has been proposed. This programming model facilitates the development of parallel and distributed computations, by providing a simple abstraction that allows the programmers to implement their algorithms without having to deal with many problems that arise in a distributed environment. This abstraction is based on two functions from functional programming languages – map and reduce. With these simple functions, the library is capable of adding all the necessary logic to perform data distribution, fault tolerance and load balancing, and the programmer is capable of developing parallel and distributed algorithms in a much more simple way that would otherwise be very hard to implement.

The simplicity of the programming model and its easiness to deploy on a cloud environment without a great amount of effort, led to a burst of applications not only in the DM-ML field but in many other fields like gene processing, log processing, economics, document clustering, amongst others.

In this document, we will propose a new algorithm called MRID4, which is heavily based on the ID3 algorithm. We will implement it using the MapReduce programming model. This will allow it to scale up to datasets with very large sizes.

Implementing an algorithm using MapReduce is not easy. The difficulty arises from the fact that the programmer still needs to deal with some of the aspects that come in a distributed environment. These aspects range from how to devise a divide and conquer implementation of the algorithm that works efficiently on MapReduce, to, how to make sure that the computed data in the mapper is somehow aggregable in the reducer.

The rest of the document is organized as follows. Chapter 2 presents the basic concepts, giving an introduction to the MapReduce programming model. Chapter 3 presents a literature review with an overview of some of the DM-ML being applied under MapReduce. Chapter 4 introduces the ID3 algorithm, and illustrates the issues with implementing it under MapReduce. Chapter 5 explains the proposed algorithm, the MRID4, leveraging a toy example to give a simple step by step explanation of its inner workings. Chapter 6 describes important implementation details and choices. Chapter 7 show how the data, used to validate the algorithm, was obtained, and the results we got from using it. And finally chapter 8 presents a conclusion of the work, and lists possible future improvements to the algorithm.

Chapter II BASIC CONCEPTS

1 CLOUD COMPUTING

The term Cloud Computing is an excessively general term, and as Luis M. Vaquero *et al.* concludes (Vaquero, Rodero-Merino, Caceresand, & Lindner, 2009): “Clouds do not have a clear and complete definition in the literature yet”. So in order to explain the concept, we will start first by introducing the analogy that Buyya *et al.* presented (Buyya, Yeo, Venugopal, Broberg, & Brandic, 2009).

In today’s evolved civilization, everyone uses utility services like water or electricity, expecting them to be readily available on demand, easy to obtain, and mostly without any knowledge of the underlying hosting infrastructure. The Cloud paradigm aims to achieve the same functionality but for a wide range of computing resources, transforming them into utility services, with the only exception that in cloud services we must share our precious data, whereas in utility services we don’t.

These resources are delivered as a service over a network, and can go from simple applications, like Google Drive (Google, 2013), or very sophisticated services, like starting and stopping individual hardware resources, network topology setup and capacity configuration or even access to a hypervisor that lets you install and manage the operation system or set of OSs of your preferred choice (Lenk, Klems, Nimis, Tai, & Sandholm, 2009).

2 MAPREDUCE

When an individual is confronted with the necessity to process huge amounts of data, in the order of many Gigabytes or even Terabytes, there is a necessity to use a lot of computers, otherwise the processing program may take an unpractical amount of time to finish. This approach means implementing a system that is able to distribute the data through all the computers, process the data in each computer and finally retrieve the processed data, optionally performing some kind of aggregation on the data delivered by each computer. The problem is that the code needed to implement this kind of distributed functionally easily subdues the original code, that is, the code that performs the actual processing becomes very small compared to the entire code. This leads to a shift in the programmer attention from the processing code to the code dealing with the data distribution, fault tolerance, load balancing and other distribution concerns. Even more, although the programmer may be an expert in the processing algorithms he most probably is not an expert in the distribution algorithms, which may lead to a bad design, longer development time and more errors.

This difficulty in creating programs capable of processing large amounts of data led Google to create, in 2004, the MapReduce programming model (Dean & Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, 2004). One of its main objectives was to allow the programmer to easily implement the processing algorithm without having to deal with all the distribution concerns, and simultaneously have a functional application that is capable of scaling.

In order to achieve such scalability and easiness, the programming model restrains the development of the application to two user defined functions: map and reduce, this abstraction allows for an easy parallelization and the use of re-execution as the primary mechanism for fault tolerance.

2.1 THE PROGRAMMING MODEL

A computation in the MapReduce programming model is defined as a linear execution of jobs, also called rounds, on a given input, usually text. The output from a job is used as the input for the next job. Each job is comprised by the execution of a user defined map function, commonly called mapper, followed by the execution of a user defined reduce function, commonly called reducer. These functions may vary in each job, such that each job performs a different transformation on the data.

The input is interpreted as a set of logical records of key/value pairs and the map function is applied to each record in order to compute a set of intermediate key/value pairs. The MapReduce library then sorts the intermediate data by key, groups all the values associated with the same key in a list and invokes the reduce function for each key and its associated list of values. The reduce function is used to combine the derived data appropriately, normally performing an aggregation on the data or simply copying the intermediate data to the output.

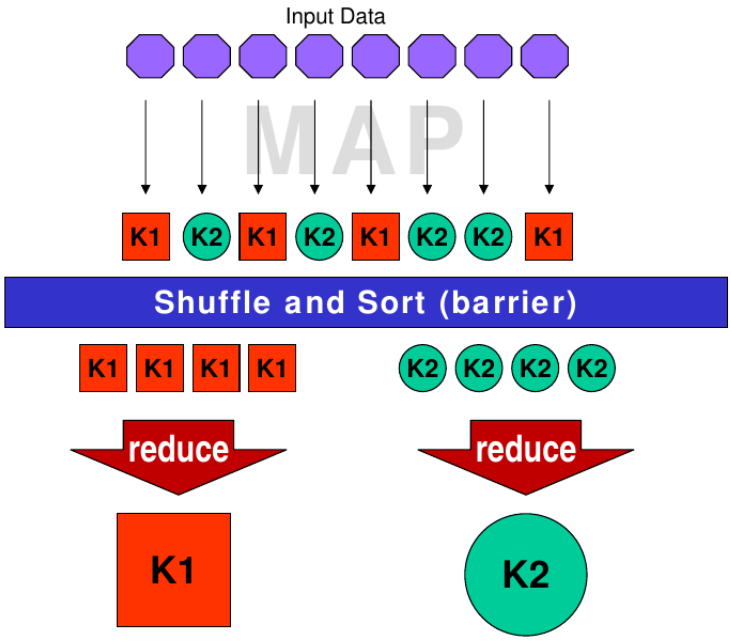


Figure 1 – An overview of the Map Reduce programming model.

The whole process is very succinctly depicted by Figure 1.

This division by key allows the library to transparently run all the map function calls in multiple machines concurrently. This is achieved by automatically partitioning the input data into a set of *M* splits and process each split in parallel by different machines. The reduce function calls can also be run in parallel by partitioning the intermediate key space into *R* pieces using a partitioning function,

that by default is *hash(key) mod R*. The user can control the partitioning by defining explicit values for the *M* and *R* variables as well as defining a custom partitioning function. The library also adds the necessary code to deal with fault-tolerance, data distribution and load balancing, freeing the user from this burden.

In addition to the map and the reduce functions there is a combine function. This function is executed after the map finishes and before the reduce starts, on the same machine that executed the map. Its main goal is to reduce the size of the data that is to be sent over the network to the reduce task. This not only improves the data transfer to the machine that will perform the reduce task, but also diminishes the time the library takes to perform the shuffle and sort because the total amount of data is less than before. In most cases, the reduce and combine functions use the same code since its effect is the same. However the library treats these functions differently, since the output of the reduce is written to the final output file, whereas the output of the combine is written to an intermediate file that will be sent to the reduce task.

In order to make the input data available at the processing node, MapReduce runs on top of a distributed file system, the Google file system (GFS) (Ghemawat, Gobioff, & Leung, 2003). The GFS distributes the data in pieces of a fixed size among the available nodes, and makes more than one copy per piece. The library takes advantage of this fact by choosing the processing node based on the data available at that node, which speeds up the processing time. The extra copies exist to make the system fault-tolerant. If a node fails, then there is a guarantee that another node will have the same data and be able to restart the failed operation.

2.2 CRITICISMS AND IMPROVEMENTS

MapReduce has been criticized from the database community (Stonebraker, et al., 2010) which claims MapReduce is better suited to perform extract, transform and load operations rather than traditional database operations. They argue for the better performance of parallel database management systems and state that MapReduce should be viewed as a complement to them, rather than a competitor. Also they criticize the fact that MapReduce is a very low level interface which conditions the ability to develop complex systems. This attack prompted the creators of MapReduce to respond with a paper showing improvements that can be made to MapReduce, that take it to a level of parallel DBMSs (Dean & Ghemawat, MapReduce: a flexible data processing tool, 2010).

In a study by Jalika *et al.* (Ekanayake & Fox, 2010) they showed that applications requiring complex communication patterns still reap a better performance using MPI style implementations over the implementations offered by MapReduce. However they also state that “handling large data sets using cloud technologies on cloud resources is an area that needs more research”. MPI, short for Message Passing Interface, is a language-independent communications protocol used to program parallel computers. Its goals are high performance, scalability, and portability. MPI is *de facto* standard for communication among processes that model a parallel program running on a distributed memory system.

Tyson Condie *et al.* shows how to enhance MapReduce in order to support pipelining within a job, that is, to be able to start the execution of reducers even if all mappers haven't finished yet, and pipelining between jobs, that is, to be able to start the next job even if all the reducers haven't finished yet (Condie, et al., 2010). These kinds of improvements are very challenging and require astute solutions.

A much more detailed discussion on possible improvements to MapReduce can be found in (Lee, Lee, Choi, Chung, & Moon, 2011).

3 DATA MINING

Data mining is the process of discovering interesting and useful patterns and relationships in large volumes of data, in such a way that the resulting structure is valued enough to further be used as input to other applications. The name is a very well chosen one since it refers us to the activity of mining, where a large amount of individuals puts a lot of effort into finding valuable minerals and ores. The main effort comes from the necessity to sift through a lot of invaluable material, such as dirt or sand, to find the precious gems. In much the same way there is normally a high cost, a computational cost, to find the most useful patterns. This cost tends to increase as the number of available data increases.

This field combines tools from artificial intelligence, statistics, machine learning and database systems in order to perform three main classes of tasks: association rule learning, clustering and classification.

3.1 COMMON TASKS

Association rule learning is the most popular method for discovering interesting relations between variables in the data set. This type of data mining is useful for tracing causal relationships in the data, such as customer purchase habits, students' choice of subjects given their grades and curricular topic interest on previous subjects.

Clustering is the task of grouping objects in the data set that share, by some measure function, a similarity. Each group is called a cluster and objects in the same cluster must be more similar to each other than to those in other clusters.

Classification is the task concerned with labeling new observations into a set of finite categories, based on a previous set of data, called a training set, containing observations whose category label is already known. Applications of this technique include for example classifying a tumor as benign or cancerous, or assigning a given email into spam or non-spam categories.

3.2 RESULTS VALIDATION

To validate the results obtained by the data mining algorithms, we have to verify that the found patterns occur in the wider data set. The most common method to achieve this is to divide the given data into two sets: the training set and the test set. The algorithm is trained on the training set, but it is tested on the test set. The resulting output of the test set is compared with the desired output, allowing

to measure the accuracy of the learned model from how many of the found patterns are in accordance with the desired output. This allows overcoming overfitting, which is the case where the patterns found in the training set are not present in the wider data set.

There are various methods to divide the data: the holdout method – the data is randomly partitioned into the two sets; random sampling – the holdout method is repeated *k* times, and the accuracy is the average of the obtained accuracies: cross-validation – the data is randomly partitioned into *k* mutually exclusive subsets, each one approximately with the same size, the algorithm is applied *k* times where at the *i*-th iteration the data set D_i is used as the test set and the others as the training set.

Chapter III RELATED WORK

In 2006 Cheng-Tao Chu et al. (Chu, et al., 2006) showed that many machine learning and data-mining (ML-DM) algorithms are suited to be implemented in MapReduce, these algorithms include *k*-Means, Naive Bayes, Neural Network, Expectation Maximization, Support Vector Machines, Logistic Regression, Locally Weighted Linear Regression, Gaussian Discriminative Analysis, Principal Components Analysis and Independent Component Analysis. Their approach to the problem consists in defining each of the algorithms as a summation over data points which may be easily parallelized in MapReduce (MR).

1 EXISTING FRAMEWORKS AND LIBRARIES

Here we present a group of already existing frameworks and libraries that aim to improve the development of machine learning and data mining algorithms. The focus is on development on top of the MapReduce paradigm except for the Weka library.

1.1 WEKA

WEKA, short for Waikato Environment for Knowledge Analysis (Hall, et al., 2009), is a data mining software that aims to make machine learning techniques generally available and to provide a unified workbench to allow researchers easy access to state-of-the-art in machine learning. It contains a collection of open sourced machine learning algorithms, and allows researchers to easily implement new algorithms without having to be concerned with supporting infrastructure for data manipulation and scheme evaluation. It is widely used for data mining research both in academia and business circles.

1.2 HADOOP

The Google's MapReduce implementation it is one of their few proprietary software's. Given its great potential there was a need for a free and open version. By 2007 Doug Cutting along with Michael J. Cafarella was able to achieve a working implementation of the programming model which they called Hadoop (Apache, 2013). Hadoop is a framework, and like Google's MapReduce, also operates on top of a distributed file system called the Hadoop Distributed File System (HDFS) (Shvachko, Kuang, Radia, & Chansler, 2010). Hadoop is fully implemented in Java, and is used by more than 190 universities/companies worldwide (Apache).

1.3 NIMBLE

Amol Ghoting *et al.* developed NIMBLE (Ghoting, Kambadur, Pednault, & Kannan, 2011), an infrastructure whose programming abstractions have been designed with the intention of parallelizing machine learning and data mining computations. Its goal is to enable rapid development of parallel ML-DM algorithms that run portably on distributed and shared-memory machines. NIMBLE is built

using a layered architecture, which makes it possible to run on top of various runtimes, such as Hadoop or MPI. Currently only Hadoop is supported.

NIMBLE defines a hierarchy of tasks. A task represents an independent, reusable piece of computation. To express an algorithm the user must implement one or more tasks. There are two kinds of tasks: Data tasks that have at least one input dataset and can generate one or more output datasets; and Non-Data tasks that represent pure computations and have neither input nor output datasets. The most useful task is the `AbstractIterativeTask` which provides a very simple abstraction for the implementation of iterative algorithms that run on top of the Hadoop platform. NIMBLE also supports task chaining which permits the creation of DAGs of tasks, that is, an explicit a priori definition of task dependencies. By making the dependencies explicit the system can co-schedule many of these chained tasks inside a single MapReduce job, thereby minimizing the number of I/O scans and the overheads of starting MapReduce jobs.

Currently NIMBLE is fully implemented and is being used by programmers at IBM Corporation, but unfortunately it isn't available to the public yet.

1.4 APACHE MAHOUT

Apache Mahout (Apache, 2013) is an Apache project to produce free implementations of distributed or otherwise scalable machine learning algorithms on the Hadoop platform. Their core algorithms for clustering, classification and batch based collaborative filtering are implemented using the MapReduce paradigm. However, contributions that run on a single node or on a non-Hadoop cluster are welcome to the project as well. There is already a great number of implemented algorithms (Apache, 2013), but there are still several algorithms missing. On the list of implemented algorithms we can find *k*-Means, Canopy Clustering, Expectation Maximization, Neural Networks, Naïve Bayes, Support Vector Machines, Parallel Frequent Pattern Mining and others.

2 ML-DM ALGORITHMS IN MAPREDUCE

Some of the classical ML-DM algorithms have been implemented using the MapReduce programming model, as seen before. In this section we present a detailed description of some of these works.

2.1 *K*-MEANS/LLOYD'S ALGORITHM

The *k*-Means algorithm (MacQueen, 1967), also called Lloyd's algorithm in computer science, first partitions the observations into *k* initial clusters, either at random or using some heuristic. Then iteratively calculates the average point, or centroid, of each cluster via some metric, and updates the clusters by moving the observations that are now closer to a different centroid. The algorithm stops when the observations no long switch clusters (or the centroids no longer change).

This algorithm has a structure that allows for an almost embarrassingly parallel implementation under MR. The initial *k* centroids are randomly obtained from the observations. The map function calculates

the distance of each observation in its input split with the existing centroids and outputs the intermediate key/value pairs where the key is the cluster ID, a number from 1 to *k*, and the value is the observation. The outputted cluster ID corresponds to the cluster to which the observation is closer. Then the reduce function updates the centroids of each cluster. The current centroids are passed to the reduce either via the Distributed Cache, or as files in the HDFS, normally the resulting files from the previous iteration. This job is re-launched if the centroids calculated in reduce have moved from the previous iteration. This approach sometimes has some minor deviations such has using different distance functions, different methods to obtain the initial centroids or using the cluster centroid instead of its ID in the output of the map.

Hai-Guang Li *et al.* (Li, et al., 2011) uses bagging (bootstrap aggregation) to improve the stability and accuracy of the algorithm. First they divide the input training set *D* with cardinality *n* into *k* training sets D_i, also with cardinality *n*, by sampling *D* uniformly and with replacement (bootstrapping). Then for each set D_i they follow the approach defined above. Finally they ensemble each of the learned models by merging the centroids obtained for each D_i (aggregation). The merging is achieved in the map function of a MR job (the reduce is the identity function), by finding a vector of centroids with minimum inner distance. The inner distance is the sum of the distances between each centroid in the vector and the centroid of the vector.

ZhenhuaLv *et al.* (Lv, et al., 2010) implemented the *k*-Means to cluster remote sensing images. Their goal is to cluster pixels that have similar colors. The implementation uses two MR jobs. The first job is responsible for assigning each pixel to a cluster and for calculating the centroids of each cluster. This is performed in the map, the reduce is the identity function. The initial centroids are defined by the user. The generated centroids in this job are merged by taking a mean of each cluster centroid as the new centroid. The second job moves each pixel to another cluster if the error variance decreases, once again this is done in the map and the reduce is the identity function. They state that the first iteration of the second job decreases the error variances immensely and that consequent iterations take trivial effects. Therefore, they only run this job once, which is the same as running the traditional *k*-Means algorithm for exactly one iteration.

Alina Ene *et al.* (Ene, Im, & Moseley, 2011) constructed a set of fast clustering algorithms in MR that use sampling to decrease the data size and run a time consuming clustering algorithm such as local search or Lloyd's algorithm on the resulting data set. These algorithms run in a constant number of MapReduce rounds.

CRITICAL REVIEW

The *k*-Means has a structure that is easily parallelized. So as expected the implementations are not very far from what one could expect. Still some authors opt to employ sampling techniques to the dataset in order to decrease the data size. Although this leads to a clear reduction in the execution time, the obtained results may be skewed.

2.2 APRIORI ALGORITHM

Apriori algorithm (Agrawal & Srikant, 1994) works iteratively, by generating candidate item sets of length *k*, called C_k, from the frequent item sets of length *k-1*, called L_{k-1}, that have been calculated in the previous iteration. The L_k is obtained by removing from C_k the candidates which do not have a minimum, user defined, support threshold. The algorithm stops when either the L_k or the C_k are an empty set, and the end result is the union of every L_k.

Lingjuan Li *et al.* (Li & Zhang, 2011) uses one MR round, where the mapper generates all the item set candidates (C_1 through C_k) for its data partition, emitting intermediate key/value pairs in the form $\langle item\ set, 1\rangle$, where the one is the support count. A combiner is used to reduce the amount of data being transferred between nodes. The reducer sums up the support counts (the values of the intermediate data) for each item set and calculates the L_k. Finally the output of each reducer is merged to obtain the final set of frequent item sets. Using this scheme the data is only scanned once.

Ning Li *et al.* (Li, Zeng, He, & Shi, 2012) uses *k* MR rounds. In the *i*-th round each item set of C_i is counted in the map. The reduce sums up the support counts and generates L_i from C_i. This implementation closely resembles the serial implementation of the algorithm with its iterative nature.

Othman Yahya *et al.* (Yahya, Hegazy, & Ezat, 2012) uses two MR rounds. The first round applies, in the map, the serial Apriori algorithm on each data partition using a partial minimum support count, which is equal to the number of transactions in the partition multiplied by the minimum support threshold. The reduce simply outputs the found partial frequent *k*-item set elements, which correspond to the global candidate frequent *k*-item sets due to the shuffle and sort phase of MR. In the second round, the map counts the occurrences of each element of the global frequent *k*-item set in the split. The global frequent *k*-item set is obtained via the distributed cache. The reduce aggregates each count in order to calculate the support count for each element in the whole data set. The output of reduce is the global support count for each element of the frequent *k*-item set. The authors of this paper claim that their implementation outperforms the *1*-round and *k*-rounds implementations; however their experiments were performed using only one node.

Ming-Yen Lin *et al.* (Lin, Lee, & Hsueh, 2012) proposes 3 algorithms. The first one called Single Pass Counting (SPC) algorithm needs *k* MR rounds, essentially the same as (Li, Zeng, He, & Shi, 2012). SPC has two key short comes: in the last few phases the scheduling of the mappers and reducers becomes an overhead compared to their workload and each round performs a scan through the entire data set, which in the case where only few candidates are counted represents a major overhead. To solve these problems they present the Fixed Passes Combined-counting (FPC) algorithm that statically combines the generation of *n* consecutive L_k into one single MR round, *n* is a fixed value in every round. This algorithm still presents a weakness since the number of generated candidates at earlier passes can be too large to be combined with candidates of longer length, which leads to the inability to prune candidates with flexibility. The Dynamic Passes Counting (DPC) algorithm solves this by dynamically adjusting *n* in each round. This is achieved by continuously generating and collecting

candidates of longer length until the total number of candidates is larger than a previous calculated threshold. This threshold is calculated as a proportion to the number of the longest frequent item sets in the last phase.

CRITICAL REVIEW

As could be expected the *1*-round implementation is very inefficient. Mainly because the item sets are only pruned at the end, in the reduce, which means a lot of data needs to be transferred between mappers and reducers just to be discarded since a lot of the generated candidates won't have the minimum support threshold. In this manner, the anti-monotonicity property is not explored, and this algorithm does much more work than the original one.

The *k*-round implementation is more attractive, because it takes more advantage from the counts distribution. However it is still not the most attractive because the overhead in the last rounds is too great compared to the computation performed.

The *2*-rounds implementation seems to be a good one based on the presented results, however given that their results were obtained using only one node, the communication costs associated with the algorithm have been greatly shielded. This is evident in their use of the distributed cache. If the experiments were to be performed on *n* nodes, the data stored in the distributed cache, that is the union of all C_k, would need to be copied to the *n* nodes. This is somewhat equivalent to what the *1*-round implementation does.

Finally the most interesting algorithm is the Dynamic Passes Counting, which is able to maintain a very good computation to communication ratio in all its rounds, which in a distributed environment represents a very well balanced system with the ability to scale up to a very high number of nodes.

2.3 DECISION TREE LEARNERS

The task of building decision tree learners in parallel is an area of many research, however there aren't many papers providing implementations for decision tree learners using MR.

Gongqing Wu *et al.* (Wu, et al., 2009) proposed the MReC4.5 algorithm, that partitions the data into *m* subsets (*m* is a user defined variable), applies the standard C4.5 algorithm in each subset (the algorithm is applied in the mapper) and finally each of the decision trees, one from each mapper, are assembled in the reducer by relying on the bagging ensemble strategy. This strategy classifies an example with class *y*, if the majority of the decision trees classify the example with class *y*.

Biswanath *et al.* created PLANET (Panda, Herbach, Basu, & Bayardo, 2009), which stands for Parallel Learner for Assembling Numerous Ensemble Trees. Their approach is based on the typical greedy top-down approach to construct each tree. At the root of the tree the entire training dataset is examined to find the best split predicate for the root. The dataset is then partitioned along the split predicate and the process is repeated recursively on the partitions to build the child nodes. However,

because of the size of the dataset and the main memory restrictions, they make use of MapReduce to be able to build the tree.

The process is as follows: a single machine, called the Controller, maintains a `ModelFile` containing the tree constructed so far and two queues: `MapReduceQueue` (MRQ) and `InMemoryQueue` (InMemQ). Whenever a dataset associated with a node is too large to fit in the main memory, that node it is added to the MRQ, similarly if the dataset fits in memory then the node is added to the InMemQ. For each node in the MRQ, a `MR_ExpandNodes` job is scheduled. This job receives the nodes to expand, N, the model file, M, and the entire training dataset. Then the map executes two actions. First, it determines if the record is part of the input dataset for any node in N, by traversing the current model with the record. Then the possible splits for each node are evaluated and the best one is selected. When a `MR_ExpandNodes` job returns, the queues are updated with new nodes that can now be expanded, if these new nodes are in the same level in the tree, PLANET will schedule a single job passing in the new nodes (as N), as opposed to scheduling a job for each node, this means the tree is expanded in a breadth first manner.

For ordered attributes (that is, continuous attributes), a split point is considered from every histogram bucket of the attribute. The histogram for each ordered attribute is computed prior to the tree induction in a MR job, and the computed histograms are approximate equi-depth histograms. The mapper maintains a table $T_{n,X}[s]$ for each node in n and for each attribute X. The keys for the table are the split points s to be considered for X, and the values are tuples (agg_tup) of the form $\{\sum y, \sum y^2, \sum 1\}$. $\sum y$ is the sum of Y values for training records (x, y) that are input to n and have values for X that are less than s, similarly $\sum y^2$ is the sum of the squares of these values. And $\sum 1$ is this number of records that are input to n and have values of X less than s. The mappers compute an agg_tup for all split points being considered for each node in N. And emit intermediate data with the keys being n, X, s and the values being the corresponding $T_{n,X}[s]$.For unordered attributes a table $T_{n,X}[v]$ is also maintained, however the keys are now the unique values of X seen in the input records to node n. As before the values of the table are tuples as already described. The emitted intermediate data is different, now the key is n, X and the value is $\langle v, T_{n,X}[v]\rangle$. In addition to the tables each mapper also maintains a tuple agg_tup_n for each node n in *N*. These tuples are also emitted to the reducers to help them compute the split qualities.

Each reducer maintains a table $S[n]$ containing the best split seen by the reducer for node n. For ordered attributes, the values received by the reducer are aggregated into agg_tup_{left}, which gives the $\{\sum y, \sum y^2, \sum 1\}$ for all records input to n that fall in the left branch of $X < s$. Using both the agg_tup_{left} and the agg_tup_n, the variance (their heuristic function) is easy to compute and update in the table $S[n]$. For unordered attributes, the values are also aggregated to get $\{\sum y, \sum y^2, \sum 1\}$ for all records to n where the value of X is v. But now the Breiman algorithm is used to find the optimal split for X.

Similarly to the nodes in the MRQ, for each node in the InMemQ a `MR_InMemory` job is scheduled, with the same arguments as the previous job. As before, in the map, the training dataset is filtered to records that are input to the node, but now it is the reducer that performs the entire sub tree construction using the traditional greedy top-down approach. It is able to do so, because all the input records fit in the main memory.

The focus of PLANET is on ordered attributes, so their implementation is more focused on regression trees.

Wei Yin et al. (Yin, Simmhan, & Prasana, 2012) implemented an open-source version of PLANET called OpenPlanet. This implementation distinguishes itself from PLANET by leveraging the Weka machine learning library.

CRITICAL REVIEW

The parallelization of decision trees is much more complicated than that of clustering or that of association rule mining, as can be easily seen by the space it took to minimally describe the PLANET. Because of this difficulty there are very few decision trees implemented under MapReduce. However their implementation is still very valuable in order to process large amounts of data.

Some authors have taken the easy road and avoided the parallelization of the tree induction. But even in this case they showed that there are benefits from these approaches.

Chapter IV ID3 UNDER MAPREDUCE

This work is being developed under the *educare* project and as such its goal is to be integrated as part of the recommender subsystem. This subsystem will be responsible for the generation of recommendations to students on which subjects to choose, given their successes and failures on previous subjects. These recommendations will be achieved through a classification algorithm, which will classify the new instances (subjects) on a categorical class label.

The work will be developed using the Hadoop framework for an easy parallelization and distribution, and it will be deployed on the RNL cluster.

First, we will develop a parallel and distributed implementation of the ID3 algorithm using the MapReduce paradigm.

1 VANILLA ID3

The ID3 algorithm (Quinlan J. R., 1986) generates a decision tree from a given dataset in a recursive manner. In the beginning the algorithm starts with the original dataset $\mathbb{S}$ as the root node, and then in each step it calculates the information gain (IG) of every unused attribute in the set $\mathbb{S}$. It then splits the set into subsets using the attribute for which the information gain is maximal; finally, it recurses on the subsets considering only the attributes never selected before. The algorithm stops the recursion on the following cases:

1. Every instance in the subset belongs the same class. A leaf labeled with the class of the instances is returned.
2. There are no more attributes to be selected, but the instances still do not belong to the same class. A leaf labeled with the most common class in the subset is returned.
3. There are no more instances in the subset. A leaf labeled with the most common class of the instances in the parent set is returned.

Instead of the IG, the normalized information gain (Quinlan J. R., 1993), or the gini-index (Breiman, Friedman, Stone, & Olshen, 1984) may be used.

The pseudo code for the ID3 algorithm is presented below.

```
 1 Function ID3 (Instances, TargetAttribute, Attributes)
 2     Root = new tree node
 3     If all instances have the same class label C Then #Stop condition #1
 4         Root.Label = C
 5     Else If Attributes is empty Then #Stop condition #2
 6         Root.Label = most common value of the TargetAttribute in Instances
 7     Else
 8         For Each attribute A in Attributes
 9            Calculate the Information Gain in Instances for the attribute A
10         End For
11         A_Best = the attribute with the highest Information Gain
12         Root.Attribute = A_Best
13         For Each possible value Vi in A_Best
14             Branch = new tree branch corresponding to the test A_Best = Vi
15             Add Branch below Root
16             Instances(Vi) = subset of instances that have the value Vi for A_Best
17             If Instances(Vi) is empty Then #Stop condition #3
18                 Node = new tree node
19                 Node.Label = most common value of the TargetAttribute in Instances
20                 Add Node below Branch
21             Else
22                 Node = ID3 (Instances(Vi), TargetAttribute, Attributes – {A_Best})
23                 Add Node below Branch
24             End If
25         End For
26     End If
27     Return Root
29 End Function
```

Algorithm 1 – The pseudo-code for the vanilla ID3 code

Although this is not evident from the pseudo-code, the time consuming part of the algorithm is iterating through the instances to compute the IG for each attribute (lines 8-10).

Also note that in line 16 the initial dataset of instances is subdivided into *n* subsets, where *n* is equal to the number of different values of the `A_Best` attribute, and the algorithm is then recursively invoked on these smaller subsets. If these subsets were to be built in-memory, the main memory would be entirely filled. Similarly writing these new subsets into disk would occupy a large amount of space, and would take a lot of time since writing to disk is a costly operation.

As a last note, we would also like to make a remark on the stop condition number 1, which is the one that verifies if all the instances have the same class label. It is easy to see this requires a full iteration of the instances under consideration in the step.

In the next section we will go through a list of issues that arise when ID3 is to be implemented under MapReduce. We will first give a brief recap of the workings of MapReduce and then present the multiple issues taking into consideration the details presented above. The next section will also serve as a stage to present the challenges that appear when developing an algorithm in MapReduce.

2 ISSUES WITH IMPLEMENTING ID3 UNDER MAPREDUCE

A MapReduce program consists in the execution of various jobs. Each job is comprised by the execution of two user-defined functions: the *Map* and the *Reduce*, there is also a possible third function, the *Combine*. In MapReduce the input is interpreted as a set of key/value pairs and the Map

and Reduce functions perform operations over them. At the beginning of each job the framework splits the input dataset into independent chunks, called *splits*, which are then processed by the mappers in a completely parallel manner. Each mapper, that is, the execution of the user defined map function over one of splits, will emit an intermediate set of key/value pairs. This intermediate data will be sorted by key by the framework and then passed as input to the reducers. Optionally before the data is sorted and sent via the network to the reducers, the user defined combine function may be invoked in order to perform an early aggregation of the intermediate data and therefore decrease the time needed to send the data over the network. The reducers will perform the final aggregation of the data and/or any other type of data processing procedure the algorithm may require. Please note that, for each unique key in the intermediate data, the framework will only assign a single reducer to process all the values associated with that key. Each reducer can run in parallel because it will be processing a different portion of the intermediate data. But the reducers can only begin to execute when all the maps have finished and the framework has sorted all the intermediate data.

2.1 1ST ISSUE – SUBSET ITERATION AND COMPUTATION

In here, we will show that mimicking the recursive implementation of the algorithm, that is, to compute the subsets and only scan those subsets in each job, is immensely costly.

Let's assume that we had a MapReduce job, *J*, which given a dataset, $\mathbb{S}$, would be able to compute the attribute that best splits $\mathbb{S}$, and the subsets resulting from splitting $\mathbb{S}$ using that attribute. With this job, we could replicate the recursive implementation of the algorithm, by invoking *J* for the subsets returned by the previous execution of *J*. Although this seems like a good solution to the problem we will show that it has a lot of shortcomings. To do this, we will first describe why the recursive execution of *J* for each sub set is not a good option, and then explain why the job *J* cannot exist.

If we were to iterate through a big dataset with MapReduce, the framework would divide the dataset, and each mapper would only have access to a split of the dataset. So, if one of the mappers finds that all its instances have the same class label (stop condition #1), this might not be true globally since one of the other mappers might have found that the instances in its split do not belong to the same class. This division also complicates the computation of the IG for each attribute, as explained in the next issue. Also, if the computed sub sets in a previous execution of *J*, were to have very different sizes, the invocations of *J* for the very small sub sets would do very little work compared to the invocations of *J* for the very large sub sets, or in other words it would lead to an uneven load distribution.

Now it should be clear that invoking *J* for each computed subset would be a bad option. Now let's show that such job *J* cannot be implemented in Hadoop. *J* needs to do two different things: to compute the attribute that best splits the dataset and to compute the sub sets resulting from splitting the data set using that attribute. Since the job must output the subsets in the end, the reducer(s) must have access to the sub sets instances, which in turn means the maps have to send the instances to the reducer(s). Even without delving deeper one can see this will incur in lots of overheads since the

entire data set has to be sent through the network and then written to a distributed file system, the HDFS.

If we were to assume the number of reducers, in *J*, is equal to the number of sub sets, that is, each reducer would output a single sub set of the data that would mean the mappers would be capable of computing the sub sets by themselves, and the reducers would only copy the input they received to the output. This however would be impossible because it would mean each mapper would have to choose the same attribute as the one that best splits the data set, but since each mapper only has access to a split of the data set, this would be extremely unlikely (we present an example of this in the next issue).

Assuming the mappers by themselves cannot compute the attribute that best splits the dataset (which we will show in the next issue). The entire data set would have to be sent to a single reducer which would then compute the sub sets, but this is the same as executing the algorithm in a single machine although with a lot of overhead.

Since sending the instances between jobs is clearly a very costly option, one of the best options seems to be to iterate the entire data set in each job, and for each example transverse the decision tree constructed so far, to see if the example falls in a node we are trying to compute the best attribute. Although this also seems a costly approach we will show that it is in fact a very good one in the 4th issue.

2.2 2ND ISSUE – COMPUTING THE IG

In here, we will show that we cannot simply compute the IG for each split and then sum the intermediate values in the reduce, so that we can pick the attribute with the highest IG.

Given that the time consuming part of the algorithm is iterating thought the instances to compute the IG for each attribute, we could leverage the MapReduce framework to distribute this work. We could launch a job where each map would calculate, for its split, the information gain for each attribute, then the reduce would sum the partial IG values and choose the attribute with the highest information gain.

Unfortunately this is not possible since $IG(\mathbb{S}+\mathbb{T},a) \neq IG(\mathbb{S},a)+IG(\mathbb{T},a)$, here $\mathbb{S}+\mathbb{T}$ denotes a set that is the union of the set $\mathbb{S}$ and the set $\mathbb{T}$ and a denotes an attribute. To prove this let us show an example, for this we will first introduce some formal definitions.

FORMAL DEFINITIONS

Let $\mathbb{S}$ be a set of instances with cardinality $|\mathbb{S}|$, and $attr(\mathbb{S})$ the set of attributes describing the instances. For an attribute $a \in attr(\mathbb{S})$, $vals(a)$ represents the set of its possible values.

A training set $\mathbb{T}$, is a set of instances of the form $(\mathbb{x}, y) = (x_1, x_2, \dots, x_n, y)$, where $x_i \in vals(a_i)$ is the value of the *i*-th attribute and $y \in \mathbb{C}$, with $\mathbb{C}$ the set of possible values for the class label. Additionally,

consider that $\mathbb{T}_{x_i=v}$ is a subset of $\mathbb{T}$, where the *i*-th attribute has value $v \in vals(a_i)$, and $\mathbb{T}_{y=c}$ a subset of $\mathbb{T}$ where the class label has value $c \in \mathbb{C}$.

$\mathbb{T}$	The set of training instances
$attr(\mathbb{S})$	The set of all attributes in a given set $\mathbb{S}$
$vals(a)$	The set of possible values for the attribute $a \in attr(\mathbb{S})$
$\mathbb{C}$	The set of possible values for the class label
$(\mathbb{x}, y) = (x_1, x_2, \dots, x_n, y)$	A training example, where $x_i \in vals(a_i)$ is the value of the i-th attribute and $y \in \mathbb{C}$ is the value of the class label
$\mathbb{T}_{x_i=v}$	A subset of $\mathbb{T}$ where the i-th attribute has value $v \in vals(a_i)$
$\mathbb{T}_{y=c}$	A subset of $\mathbb{T}$ where the class label has value c $\in \mathbb{C}$
$\lvert\mathbb{S}\rvert$	The cardinality of a set $\mathbb{S}$

Table 1 – Summary of the formal definitions

$$H(\mathbb{S}) = -\sum_{c \in \mathbb{C}} \frac{|\mathbb{S}_{y=c}|}{|\mathbb{S}|} log_2 \frac{|\mathbb{S}_{y=c}|}{|\mathbb{S}|}$$

Equation 1 – The entropy of a set $\mathbb{S} \subseteq \mathbb{T}$

$$IG(\mathbb{S}, a) = H(\mathbb{S}) - \sum_{v \in vals(a)} \frac{|\mathbb{S}_{x_a=v}|}{|\mathbb{S}|} H(\mathbb{S}_{x_a=v})$$

Equation 2 – The information gain in the set $\mathbb{S} \subseteq \mathbb{T}$ for an attribute $a \in attr(\mathbb{S})$

Record Number	Blood Pressure	Obesity	Smokes	Stroke
1	High	Yes	No	Yes
2	Low	No	Yes	No
3	Low	Yes	Yes	Yes
4	Normal	Yes	No	Yes
5	High	Yes	Yes	Yes
6	Low	No	No	No
7	Normal	No	Yes	Yes
8	Normal	Yes	Yes	Yes
9	Normal	No	No	No
10	High	No	No	No
11	High	No	Yes	Yes
12	Low	Yes	No	No

Table 2 – A simple toy example.

If we consider that the set $\mathbb{S}$ is defined by the records 1 through 6 inclusive, the set $\mathbb{T}$ is defined by the remaining records, and that we are computing the IG for the Smokes attribute, we will obtain the following values:

$$IG(\mathbb{S}, Smokes) = -\frac{4}{6}log_2\frac{4}{6} - \frac{2}{6}log_2\frac{2}{6} - \frac{3}{6}\left(-\frac{2}{3}log_2\frac{2}{3} - \frac{1}{3}log_2\frac{1}{3}\right) - \frac{3}{6}\left(-\frac{2}{3}log_2\frac{2}{3} - \frac{1}{3}log_2\frac{1}{3}\right) = 0$$

$$IG(\mathbb{T}, Smokes) = -\frac{3}{6}log_2\frac{3}{6} - \frac{3}{6}log_2\frac{3}{6} - \frac{3}{6}\left(-\frac{3}{3}log_2\frac{3}{3}\right) - \frac{3}{6}\left(-\frac{3}{3}log_2\frac{3}{3}\right) = 1$$

$$IG(\mathbb{S}+\mathbb{T}, Smokes) = -\frac{7}{12}log_2\frac{7}{12} - \frac{5}{12}log_2\frac{5}{12} - \frac{6}{12}\left(-\frac{5}{6}log_2\frac{5}{6} - \frac{1}{6}log_2\frac{1}{6}\right) - \frac{6}{12}\left(-\frac{2}{6}log_2\frac{2}{6} - \frac{4}{6}log_2\frac{4}{6}\right)$$
$$= 0.195709$$

This proves that $IG(\mathbb{S}+\mathbb{T}, a)$ is not necessarily equal to $IG(\mathbb{S}, a) + IG(\mathbb{T}, a)$, which means we cannot simply calculate the IG in each split and add the partial IG values in the reduce. We can however calculate intermediate values in each mapper and aggregate them in the reduce, in such a way that the aggregated values are equivalent to the IG. We will show the reasoning of this in a later section.

2.3 3RD ISSUE – ONE ATTRIBUTE PER MAPPER

In here, we will show that parallelizing the algorithm in a column-wise approach is expensive to implement in MapReduce. Additionally, this approach would have some scalability problems.

It seems reasonable to distribute the work of computing the IG for each attribute in a column-wise approach, that is each mapper would calculate the IG for an attribute (a column in the data set) and the reduce would choose the one with the highest value. However this distribution also lacks in various aspects.

In Hadoop, we can't reliably ascertain how many map tasks will be launched, this coupled with the fact that we cannot distribute different data to each mapper (besides the dataset split or any data in the distributed cache or job configuration), makes it so that we cannot inform each mapper what attribute it should process. Even if we could do this, each mapper would have to iterate the entire dataset to be able to compute the IG for its given attribute (as shown in the 2nd issue), however since each mapper only gets a split of the input this would be impossible.

We could still implement this approach in two different ways. The first one would be to launch a MapReduce job for each attribute. This way, we would iterate though the entire dataset (we would once again find it troublesome to compute the IG). The second approach would be to do some cleaver tricks on how the framework launches map tasks, and make it so that the framework would launch a map task for each attribute in the dataset, instead of distributing the dataset, we would be distributing upon a file with an attribute per line and launching a map task for each line in the file. This way each mapper could read the entire dataset from the HDFS and compute the IG for the given attribute.

As should be clear, either alternative is very heavy since it requires that the entire dataset is iterated for each existing attribute. And the second approach is even more problematic since is restrains the number of working machines to the number of attributes, which in a big cluster would mean most of the machines would be idle.

Here the best approach is to calculate the IG for every attribute simultaneously in each mapper of the job. As explained in the previous issue, the computed value will not be the IG itself but some value that allows for the computation of the IG in the reduce.

2.4 4TH ISSUE – WHEN TO LAUNCH A MAPREDUCE JOB

In here, we will show that launching a job for each decision tree node is an expensive approach.

Here the discussion is when should we launch a job and what should it be calculating. This is a very pertinent question since launching a job has considerable setup times (overheads). If we launch too many jobs, we will lose a lot of time setting up each one, but if we launch too few jobs we will lose flexibility in the system, because each job will have to perform much more work.

Although we haven't yet explained how, assume that we have a MapReduce job that iterates through the entire dataset and is able to compute the attribute that best splits the dataset under a given node. This job determines if an instance falls within a given node by traversing the decision tree computed so far.

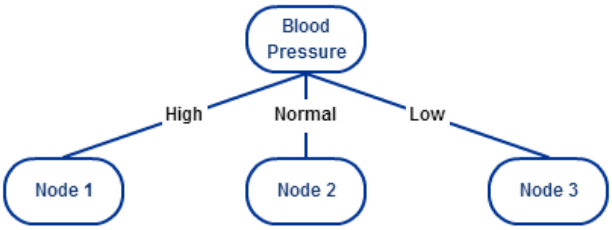


Figure 2 – An incomplete decision tree.

Following the previously presented training dataset, assume the mentioned job ascertained that the Blood Pressure is the best attribute for the root of the decision tree, and we are now faced with the choice of how to calculate the best attribute for each of the three child nodes of the Blood Pressure attribute node, as shown in Figure 2. We could do this by launching the job for each of the child nodes or we could launch the job for the three child nodes simultaneously, that is, this single job would compute the best attribute for each node of an entire level of the decision tree. Although not obvious let us demonstrate that the best option is to launch a single job. To do this we will explain what would happen if we were to launch a job for each node.

We know that each job we must iterate the entire data set. So, in the job relative to the node 1, we would have to iterate through the instances where the Blood Pressure is Normal and Low, only to find that these instances do not fall within Node 1. Which means all the work needed to traverse the decision tree for each of these instances would be wasted. And because we are launching three jobs we would be wasting even more time because of the setup time of each job, as well as the unnecessary iteration of the entire data set thrice.

If we instead launch a single job for every level of the tree, we will have the guarantee that every time we traverse the decision tree the example will always fall within a node we are interested, and

therefore we will not waste any computation. With this approach we are trading computation for storage, but since the dataset is assumed to be big we will clearly reap a greater benefit with this strategy.

Chapter V MRID4

As should have become apparent in the previous chapter, the problem of implementing ID3 in MapReduce comes down to: how to compute the IG for each attribute. We showed this is no trivial task and gave some hints on how to solve the problem; however we did not explained how. In this chapter we will explain how to overcome the presented issues, and explain how our solution will operate, to do so we will first introduce the concept we call Class Counts which is the core of our algorithm.

MRID4 stands for **M**ap**R**educe **I**terative **D**ichotomizer **4**. The number 4 was chosen because our algorithm also supports continuous attributes as proposed in the C4.5 algorithm but it is not a MapReduce implementation of the C4.5 algorithm. In other words, it sits between the ID3 algorithm and the C4.5 algorithm.

1 CLASS COUNTS

This concept overcomes the very restrictive equation $IG(\mathbb{S}+\mathbb{T},a) \neq IG(\mathbb{S},a)+IG(\mathbb{T},a)$, and allows an easy parallelization of the IG computation. We will make use of the formal definitions introduced in the previous chapter to explain it.

The first step is to take Equation 1 and Equation 2 and manipulate them into a more manageable representation.

$$H(\mathbb{S}) = -\sum_{c \in \mathbb{C}} \frac{|\mathbb{S}_{y=c}|}{|\mathbb{S}|} log_2 \frac{|\mathbb{S}_{y=c}|}{|\mathbb{S}|} = \frac{|\mathbb{S}| \log|\mathbb{S}| - \sum_{c \in \mathbb{C}} |\mathbb{S}_{y=c}| \log|\mathbb{S}_{y=c}|}{|\mathbb{S}| \log 2}$$

Equation 3 – A more manageable representation of the entropy of a set.

See Appendix, point 1, for a full algebraic prof of this transformation.

Using this representation for the entropy of a set, we can write the Information Gain equation as follows.

$$IG(\mathbb{S},a) = H(\mathbb{S}) - \sum_{v \in vals(a)} \frac{|\mathbb{S}_{x_a=v}|}{|\mathbb{S}|} H(\mathbb{S}_{x_a=v})$$

$$= H(\mathbb{S}) - \sum_{v \in vals(a)} \frac{|\mathbb{S}_{x_a=v}|}{|\mathbb{S}|} \left(\frac{|\mathbb{S}_{x_a=v}| \log|\mathbb{S}_{x_a=v}| - \sum_{c \in \mathbb{C}} |\mathbb{S}_{x_a=v \wedge y=c}| \log|\mathbb{S}_{x_a=v \wedge y=c}|}{|\mathbb{S}_{x_a=v}| \log 2} \right)$$

$$= H(\mathbb{S}) - \frac{1}{|\mathbb{S}| \log 2} \sum_{v \in vals(a)} \left(|\mathbb{S}_{x_a=v}| \log|\mathbb{S}_{x_a=v}| - \sum_{c \in \mathbb{C}} |\mathbb{S}_{x_a=v \wedge y=c}| \log|\mathbb{S}_{x_a=v \wedge y=c}| \right)$$

Equation 4 – A more manageable representation of the information gain fo a set and an attribute.

In order to compute the attribute that best splits a given data set we are computing the following equation:

$$\max_{a \in attr(\mathbb{S})} IG(\mathbb{S}, a)$$

Equation 5 – Criterion for choosing the attribute that best splits the data set $\mathbb{S}$.

Note that in this equation the set $\mathbb{S}$ is always the same, this means that the values $H(\mathbb{S})$ and $|\mathbb{S}|$ in Equation 4 are always constant. Since we are not interested in knowing the actual value of the IG, but rather the attribute with the highest IG we can transform Equation 5 into a minimization.

$$\max_{a \in attr(\mathbb{S})} \left(H(\mathbb{S}) - \frac{1}{|\mathbb{S}| \log 2} \sum_{v \in vals(a)} \left(\left|\mathbb{S}_{x_a=v}\right| \log \left|\mathbb{S}_{x_a=v}\right| - \sum_{c \in \mathbb{C}} \left|\mathbb{S}_{x_a=v \wedge y=c}\right| \log \left|\mathbb{S}_{x_a=v \wedge y=c}\right| \right) \right)$$

$$= H(\mathbb{S}) - \frac{1}{|\mathbb{S}| \log 2} \left(\max_{a \in attr(\mathbb{S})} \sum_{v \in vals(a)} \left(\left|\mathbb{S}_{x_a=v}\right| \log \left|\mathbb{S}_{x_a=v}\right| - \sum_{c \in \mathbb{C}} \left|\mathbb{S}_{x_a=v \wedge y=c}\right| \log \left|\mathbb{S}_{x_a=v \wedge y=c}\right| \right) \right)$$

$$\propto \min_{a \in attr(\mathbb{S})} \sum_{v \in vals(a)} \left(\left|\mathbb{S}_{x_a=v}\right| \log \left|\mathbb{S}_{x_a=v}\right| - \sum_{c \in \mathbb{C}} \left|\mathbb{S}_{x_a=v \wedge y=c}\right| \log \left|\mathbb{S}_{x_a=v \wedge y=c}\right| \right)$$

Equation 6 – Simplified formula to calculate the attribute with maximum information gain.

Notice that since we are iterating through all the possible values for the attribute a, the expression $\sum_{v \in vals(a)} \left|\mathbb{S}_{x_a=v}\right|$ will always be equal to $|\mathbb{S}|$, similarly for a given a and v, the expression $\sum_{c \in \mathbb{C}} \left|\mathbb{S}_{x_a=v \wedge y=c}\right|$ will always be equal to $\left|\mathbb{S}_{x_a=v}\right|$. This means that if we have $\left|\mathbb{S}_{x_a=v \wedge y=c}\right|$ for each a, v and c we can compute Equation 6 and therefore compute the attribute with the highest information gain.

This type of algebraic manipulation can also be achieved for the information gain ratio. Or in another words, with just the Class Counts we can also compute the information gain ratio in the reducers. See Appendix, point 2, for similar algebraic manipulation of the information gain ratio formula.

From hence forth, whenever we refer to class counts we are referring to $\left|\mathbb{S}_{x_a=v \wedge y=c}\right|$ for each a, v and c in the set $\mathbb{S}$. This can easily be visualized as a tri-dimensional associative array (also known as a map or dictionary). The first dimension are the attributes (each a), the second dimension are the attribute values of the attribute (each v), the third dimension are the class label values (each c) and the stored values correspond to the counts $\left|\mathbb{S}_{x_a=v \wedge y=c}\right|$.

2 THE GENERAL IDEA

Having explained what are the class counts we will now explain how the algorithm operates.

There will exist a machine, the Central node, which will: schedule the execution of MapReduce jobs; assign unique IDs to the decision tree nodes that will be processed in the next job; and read the outcome of each scheduled job into the decision tree, that is, it will maintain the model learned so far. The Central node will launch a job for each level in the tree.

Each MapReduce job will operate as follows: each mapper will calculate the partial[1] class counts, for each of the nodes in the current level of the decision tree. To perform this computation the mapper will need to have access to the decision tree computed so far, which will be available through the Distributed Cache. The output of the mappers will have as key the IDs of the decision tree nodes being calculated in the job, and as values the partial class counts for those nodes.

Each reducer will read the partial class counts and compute Equation 6, in order to find the attribute with the highest IG. The output of the reducers will have as key the node ID and as value the tuple $(Attribute, Final\ Class\ Counts)$. The $Attribute$ field will allow the Central node to know what attribute was chosen for the node and the $Final\ Class\ Counts$ field corresponds to the final[2] class counts for the chosen attribute and will be used to compute whether we are at a leaf node or if there are still child nodes that need to be computed. The explanation for this is much easier to understand with a concrete example, so we will cover it in the Toy Example section.

$$Map: \langle \_, Instance \rangle \rightarrow \langle NodeID, Partial\ Class\ Counts \rangle$$
$$Reduce: \langle NodeID, list\ of\ Partial\ Class\ Counts \rangle \rightarrow \langle NodeID, \langle Attribute, Final\ Class\ Counts \rangle \rangle$$

Equation 7 – Overview of the MapReduce job launched by the Central node.
Angle brackets represent key-value pairs and underscore means we don't care about the key.

3 PSEUDO CODE

```
1  Function CentralNode()
2     decisionTree = new DecisionTree
3     nodesToProcess = new List(0)
4     While nodesToProcess has nodes
5         job = createJob(decisionTree, nodesToProcess)
6         Wait for the job to complete
7         nodesToProcess = addNewNodes(job outcome, decisionTree)
8     End While
9  End Function
```

Algorithm 2 – The pseudo-code for the central node.

The `addNewNodes` function is also responsible for the assignment of unique IDs to the decision tree nodes that will be processed in the next job.

[1] Partial in the sense that the computed class counts are only relative to the mapper split.
[2] Final in the sense that these class counts are relative to the entire dataset.

```
Class Map
    #Field holding the decision tree constructed so far
    DecisionTree decisionTree
    #Field holding the PartialClassCounts for each node being processed
    HashMap<Node, PartialClassCounts> partialClassCounts

    Function setup(context)
        decisionTree = read decision tree from DistributedCache
        nodesToProcess = read nodes to process from context (Job configuration)
        partialClassCounts = new HashMap<Node, PartialClassCount>
        For Each node in nodesToProcess
            partialClassCounts[node] = new PartialClassCount
        End For
    End Function

    Function map(key, value, context) #value is an instance from the data set
        node = decisionTree.findLeaf(value)
        nodePartialClassCounts = partialClassCounts[node]
        For Each attribute a in node.unusedAttributes
            nodePartialClassCounts[a][value[a]][value[class]]++
        End For
    End Function

    Function cleanup(context)
        For Each (node, partialClassCount) in partialClassCounts
            context.emit(node.ID, partialClassCount)
        End For
    End Function
End Class
```

Algorithm 3 – The pseudo-code for the map function.

```
#key = node.ID and values is a list of PartialClassCounts
Function Reduce.reduce(key, values, context)
    aggregatedCounts = new PartialClassCounts
    For Each partial class count C in values
        aggregatedCounts.aggregateClassCounts(C)
    End For
    context.write(key, aggregatedCounts.getBestAttributeForSplit)
End Function
```

Algorithm 4 – The pseudo-code for the reduce function.

The `getBestAttributeForSplit` returns the tuple ⟨attribute, FinalClassCount⟩.

4 TOY EXAMPLE

To better exemplify the idea, we will show how the algorithm would work on the toy example presented before. We include the toy example here for an easy reference.

Record Number	Blood Pressure	Obesity	Smokes	Stroke
1	High	Yes	No	Yes
2	Low	No	Yes	No
3	Low	Yes	Yes	Yes
4	Normal	Yes	No	Yes
5	High	Yes	Yes	Yes
6	Low	No	No	No
7	Normal	No	Yes	Yes
8	Normal	Yes	Yes	Yes
9	Normal	No	No	No
10	High	No	No	No
11	High	No	Yes	Yes
12	Low	Yes	No	No

Table 2 – A simple toy example.

For simplicity, let us assume we have only two mappers, each one getting half of the table. As stated before, the mappers will calculate the partial class counts, and then the reducer will aggregate these counts in order to pick the attribute with the maximum IG.

The mapper maintains a hash map, mapping nodes to partial class counts, partial in the sense that the computed class counts are only relative to the mapper split. Initially, the partial class counts associated with each node is filled with zeros.

For each instance in split, the mapper will traverse the decision tree constructed so far. In the beginning the tree only has an empty root node holding the node ID 0. By traversing the tree, the mapper will know in which node of the tree the instance falls, as well as, all the unused attributes in that path of the tree.

With the node ID (0 in this example) the mapper will obtain its associated PartialClassCount, and then for each unused attribute will do (line 20 of the pseudo-code):

```
partialClassCount[attributeName][attributeValue][classLabel]++
```

So, for the first record it would do:

```
partialClassCount[BloodPressure][High][Yes]++
partialClassCount[Obesity][Yes][Yes]++
partialClassCount[Smokes][No][Yes]++
```

The first mapper, the one scanning through the first half of the table, at the end will have as partial class counts the following values:

Blood Pressure

	Yes	No
Low	1	2
Normal	1	0
High	2	0

Obesity

	Yes	No
Yes	4	0
No	0	2

Smokes

	Yes	No
Yes	2	1
No	2	1

Table 3 – Partial class counts of the first mapper for the node with ID = 0.

The second mapper will have the following partial class counts.

Blood Pressure

	Yes	No
Low	0	1
Normal	2	1
High	1	1

Obesity

	Yes	No
Yes	1	1
No	2	2

Smokes

	Yes	No
Yes	3	0
No	0	3

Table 4 – Partial class counts of the second mapper for the node with ID = 0.

Given that the current level of the tree only has a node to process, we will have only one reducer, which will process the partial class counts for the node with ID = 0. The first step of this process is to aggregate the partial values in order to obtain the aggregated class counts. This is done by simply summing the values relative to the same attribute, attribute value and class label value. After the aggregation is done we will obtain the following values:

Blood Pressure

	Yes	No
Low	1	3
Normal	3	1
High	3	1

Obesity

	Yes	No
Yes	5	1
No	2	4

Smokes

	Yes	No
Yes	5	1
No	2	4

Table 5 - The aggregated class counts obtained by the reducer.

The next step is finding the attribute with the highest information gain. This is done by computing Equation 6, which is presented below for an easy reference:

$$\min_{a \in attr(\mathbb{S})} \sum_{v \in vals(a)} \left(\left|\mathbb{S}_{x_a = v}\right| \log \left|\mathbb{S}_{x_a = v}\right| - \sum_{c \in \mathbb{C}} \left|\mathbb{S}_{x_a = v \wedge y = c}\right| \log \left|\mathbb{S}_{x_a = v \wedge y = c}\right| \right)$$

Equation 6 – Simplified formula to calculate the attribute with maximum information gain.

Let's see how this would be achieved for the Blood Pressure attribute. Remember that the $|\mathbb{S}_{x_a=v}|$ value for a given v can be computed by summing $|\mathbb{S}_{x_a=v \wedge y=c}|$ for each of the class label values (summing each line of the array). We show this in the following table for an easier understanding.

	Yes	No	$\lvert\mathbb{S}_{x_a=v}\rvert$
Low	1	3	4
Normal	3	1	4
High	3	1	4

$$
\begin{aligned}
& 4\log 4 - 1\log 1 - 3\log 3 \\
+\ & 4\log 4 - 3\log 3 - 1\log 1 \\
+\ & 4\log 4 - 3\log 3 - 1\log 1 \\
= \ & 6.748021
\end{aligned}
$$

Computing the formula for the remaining attributes we would get the following value for both attributes:

$$6\log 6 - 5\log 5 - 1\log 1 + 6\log 6 - 2\log 2 - 4\log 4 = 6.522452$$

Therefore we could choose either the Obesity or the Smokes attribute as the root of the tree. Assuming we choose the Smokes attribute, the final class count sent as output by the reducer then would be:

	Yes	No
Yes	5	1
No	2	4

This final class count is very useful to determine whether the computed node has any child nodes. In this example 83% (5/6) of the instances with the value Yes for the Smokes attribute have the class label Yes. If this percentage were to be 100% we clearly would be at a leaf node where the class label is Yes (the stop condition number 1 from the pseudo-code of the vanilla ID3). In the real case, we cannot expect this percentage to reach 100% so we must define a threshold according to our needs. In our algorithm this value is, by default, 90%.

5 CONTINUOUS ATTRIBUTES EXTENSION

Given that most of the datasets we could find contained many continuous attributes we decided to extend the algorithm to be able to deal with them.

The major difference from the discrete attributes is the way the split points are considered. With discrete attributes every value of the attribute will correspond to a split point. So for example the Blood Pressure attribute of the toy example will split the data set into three sub sets. With a continuous attribute, if we were to employ the same tactic the resulting number of sub sets would be enormous. So, we turned to the proposal by Quinlan for the C4.5 algorithm (Quinlan J. R., 1993).

His proposal is as follows: for a continuous attribute being considered, its list of values, in a set $\mathbb{S}$, are sorted to give ordered values $v_1, v_2, \ldots, v_n$; every pair of adjacent values suggests a potential threshold

$\theta = \frac{v_i + v_{i+1}}{2}$ and the corresponding partitions $\mathbb{S}_{x_a \leq \theta}$ and $\mathbb{S}_{x_a > \theta}$; the threshold for which the resulting partitions have the best information gain is selected.

With discrete attributes the mappers know all the possible values for each of the discrete attributes. They leverage this fact by using a fixed sized data structure to send the class counts to the reducers. Using the same approach for continuous attributes would result in a lot of waste, since all the possible values for a continuous attribute will not be present in each split. So to resolve this issue, the mappers will send a dynamic structure to the reducers; this structure will only contain the values of the continuous attribute that are present in the mapper split. In other words, only the reduce will have access to the complete list of the values for a continuous attribute.

Apart from this, the major differences are at the implementation level, so we will explain it further in Chapter VI.

6 TWO LEVELS PER JOB EXTENSION

In the 4th issue, we made it seem that launching a single job for all the nodes in a level of the tree was a very good approach. We gave the argument that all the computations needed to find in which node does an instance falls into, were useful work. However this is not always the case: as the decision tree grows, the probability that one of the nodes becomes a leaf node becomes higher. This will mean that at least some of these computations will be wasted, since the instance will fall into a node that will have no more children, that is, a node where no further computations are necessary.

This insight combined with the fact that the iteration of the entire dataset in each job is the most time consuming operation, prompts the need to further optimize this aspect of the algorithm. Once again, our solution comes in the form of trading computation for storage, which will be achieved by merging together the computation of two levels of the tree in a single job. Since the computation of a level is dependent on the previous level, this approach will imply that some calculations will end up being discarded.

Let us explain how to compute two levels in just one job, using as an example the first two levels.

In the first level, we need to compute the information gain for each attribute, where the mappers calculate the Class Counts and the reducers aggregate them to find the attribute with the maximum IG value. In the next level, we will need to compute the IG for each of the remaining attributes in every leaf, that is, the values of the attribute chosen in the level 1. Consequently, if we want to compute the IG for the two levels in the same job, first we need to calculate the IG for every attribute (level 1), then for each value of each attribute of level 1 we need to calculate the IG for the remaining attributes (level 2).

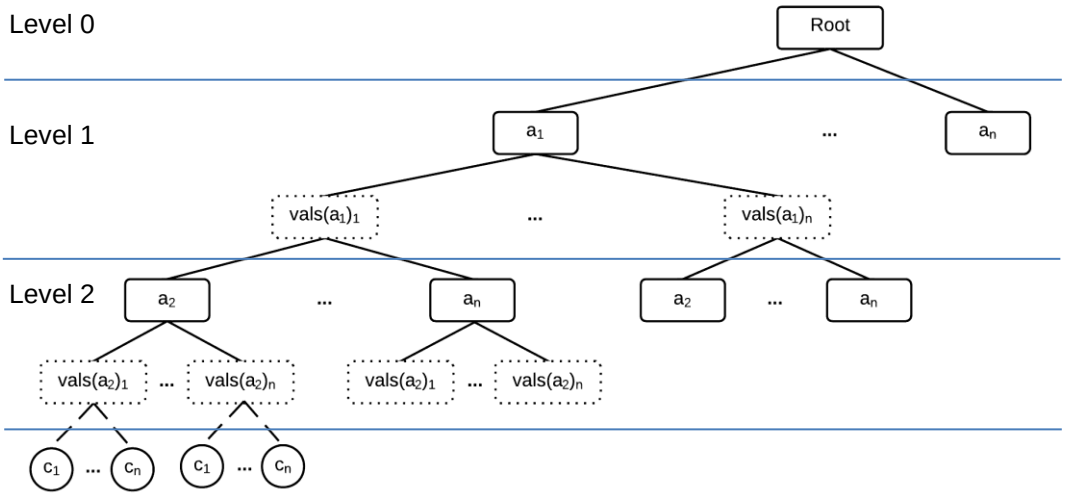


Figure 3 – Overview of the attributes and their values for two levels.

Note: the boxes represent the attributes, the dotted boxes the values and the circles the class labels.

In this scheme, we can have a reducer for each sub-tree that requires maximization. For example in level 1, we can only have one reducer; however in level 2, we can have a reducer for each sub-tree rooted in a value of the attributes of level 1. And all of these reducers will run in parallel, which will mean that in terms of time a job running for two levels of the tree will only take a little more time to run then a job for a single level. This extra time will arise from the extra data sent between the mappers and reducers and the extra time needed to perform the computations for the double level nodes in the mapper. Even though each job might take a little more time to run, we will need to run a lot less jobs (roughly half, if we always operate with two levels per job), so the overall performance of the system should be improved.

We only implement this idea for discrete attributes for two reasons: usually continuous attributes have a very wide range of different values, which would translate to a very high amount of intermediate data, and we only have access to a complete list of values for a continuous attribute in the reducer. This introduces a difficult caveat: we cannot compute the information gain for attributes on the second level when the attribute in the first level is a continuous one. In these cases we compute the two levels for all the discrete attributes in the level one. Then if the best attribute for the level one is in fact a discrete attribute, then we can immediately choose the best attribute for each of its children. However if the best attribute for the level one is a continuous attribute, then we have wasted time computing the two levels since it would be the same to compute just one level.

Chapter VI IMPLEMENTATION

In this chapter we will go thought the major implementation choices we have made.

1 NO COMBINER

The choice of not employing the use of the combine function is an easy one to explain, considering the 1st issue explained in chapter 5.

Instead of each mapper maintaining a class count for each node that it is processing, we could simply output tuples in the form $\langle NodeID, \langle AttributeName, AttributeValue, ClassLabel, 1\rangle\rangle$ and use a combiner to partially aggregate the outputted tuples. However, this would entail in a lot more overheads, since the size of the intermediate data would be much larger compared to what we have described.

2 DATA STRUCTURES FOR THE PARTIAL CLASS COUNTS

We explained that each mapper outputs tuples in the form $\langle NodeID, PartialClassCounts\rangle$, however we never explained what a $PartialClassCounts$ really is. We have been passing it off as a tri-dimensional associative array, in practice it is actually a more complex type.

For discrete attributes the $PartialClassCounts$ holds an associative array (also known as a dictionary or a map) where the keys are the attributes names and the values are arrays. The lines of these arrays correspond to the different values of the attribute, whereas the columns correspond to the different values of the class label, the cells of the array hold the count $\left|\mathbb{S}_{x_a=v \wedge y=c}\right|$. The reason for this choice comes from de fact that the number of different values for a discrete attribute, as well as for the class label (which is also discrete), is usually small, so the probability that the majority, if not all, of these values is going to be present in the split of each mapper is very high.

For the continuous attributes there is still an associative array with the keys being the attribute names, but the values are now a sorted associative array (a sorted map) with the keys being the attribute values and the values being a 1-line array with the columns corresponding to the different values of the class label as before. This difference steams from the fact that continuous attributes usually have a very high range of different values, which diminishes greatly the probability of finding the majority of them in the split of each mapper.

In addition, there is a special mapping in the second associative array (the sorted map), which maps the special value *NaN* to a 1-line array. This 1-line array also has the same columns as before (the different values of the class label), but now each position of the array holds an aggregated count of the values in the same position of all the other 1-line arrays. In other words, this special mapping holds the aggregation of all the 1-line arrays. This is very useful to be able to compute the information gain for each partition inferred by each of the potential thresholds (θ), and saves us a full iteration of the sorted

map to be able to compute the aggregation of every 1-line array. Once again, we are trading computation for storage, albeit a very little amount of storage.

3 TRANSLATION MAP

Because we are using arrays in the $Partial\ Class\ Counts$ we have to translate the class label values to indices and in the case of nominal attributes the attribute values as well. This is achieved through a translation map, which simply maps strings to integers. This translation scheme in conjunction with the arrays in the $Partial\ Class\ Counts$ has the added bonus of reducing the amount of data sent between the mappers and reducers, since we can simply send the translation map alongside with the arrays, which corresponds to a lesser amount of data, then sending the strings for each of the attribute values and class label values. Furthermore, we only need to send the translation map once.

4 TREE MODEL

The decision tree computed so far is backed by a data structure called $TreeModel$. This data structure needs to handle the fact that each node might have a different number of children; it must be easily serializable, preferably in such a way that only new nodes need to be added to the distributed file system; must be compact; and finally, it must be descriptive enough to be used as an output from the algorithm.

To achieve these properties, the chosen implementation is as follows: each tree node is a data structure serializable on its own. Each node maintains the ID of its parent node, so that we can traverse backwards on the tree, as needed for the two levels per job idea; the ID of the node; a reference to the attribute the node represents; the threshold value (θ) for continuous attributes; the final class counts for the attribute of the node, this, as explained before, is used to compute if the node has any children for each of the attribute values; and finally, a vector with as many items as the attribute values, where each item saves the ID of the children node if for that attribute value there is a child node.

Finally the $TreeModel$ simply maintains a map, mapping node IDs to tree nodes.

5 MULTI-LEVEL KEY

In Hadoop, a reducer can only receive a key of a given type, for example, an integer or string. When we are computing just one level of the decision tree per job, this is not a problem. However, when we compute two levels in a single job, this constraint becomes a bit problematic. To solve it, we need to create a super type that could accommodate both a simple key, that is, just the $NodeID$, as well as the key needed for the two levels per job idea $\langle NodeID, fromAttribute, fromAttributeValue \rangle$, which we call a *multi-level key*. Since the simple key is contained in the multi-level key, we just use the multi-level key. When we are calculating just one level of the tree, we have a very small overhead since the last two fields of the key will not be used. Unfortunately this is something we cannot change easily.

Chapter VII VALIDATION

In this chapter, we will endeavor to explain how we constructed the dataset we used to test our algorithm. And the results we got from using it.

1 THE DATA

Our work is being done under the educare project. And, as such we are required to follow the project main goals. This lead us to use an educational dataset, constructed using the data within the project's data warehouse. This warehouse is focused on IST student's data.

1.1 CONSTRUCTION OF THE DATASET

First, we collected the list of every student that enrolled in the subject Foundations of Programming, since 2006 and up to 2012 inclusive. We only considered the first semester enrollments, since we are after the complete list of freshman enrollees. Secondly, we removed from the list every student that enrolled in any of the previous years. In other words, we removed the students that failed the subject and that reenrolled in a later year. The end result is a list of students that enrolled in the degree of Information Systems and Computer Engineering each year from 2006 to 2012.

For the next step, we used two of the star schemas of the warehouse: *subject enrollments* and *subject results*.

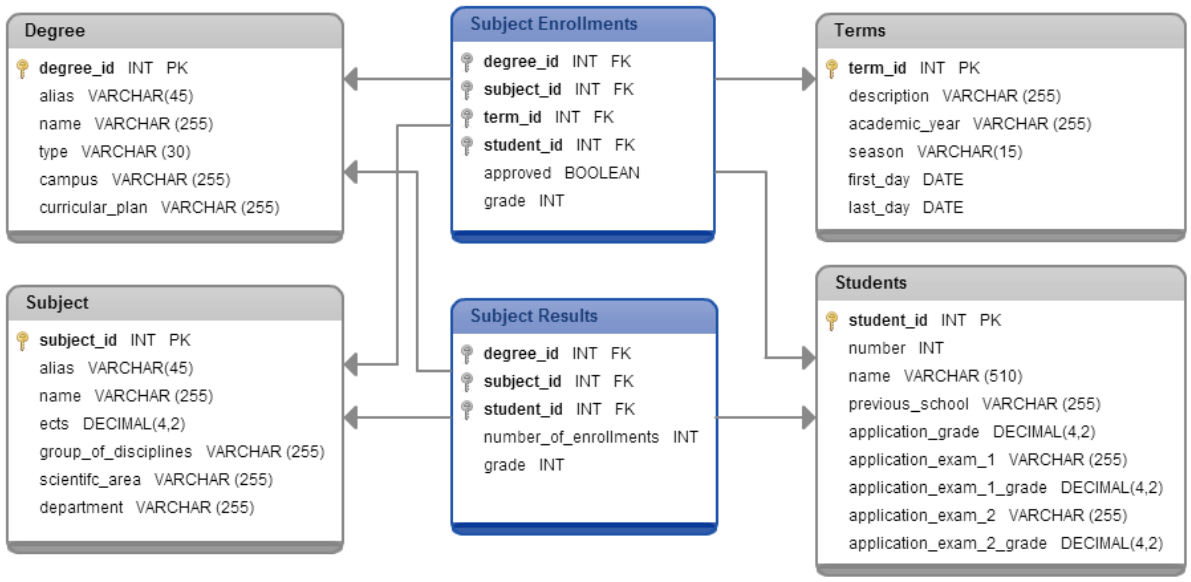


Figure 4 – The database schema for the stars subject enrollments and subject results.

The star *subject enrollments* has an entry for each enrollment a student has made in the degree, if the student has been approved in the given subject and the grade he obtained.

The star *subject results* is an aggregation of the previous star, where data is aggregated according to the time dimension, which in the educare data warehouse is represented by the term. However the aggregation is only performed if the student has been approved in a given subject. For example, the student Simão, finished the subject Foundations of Programming with grade 16 and enrolled 1 time to do so.

It is also important to note that in the data warehouse the time only spans to the second term of 2012, that is, after this term we have no data.

From these two stars, we computed a table with two columns per subject, in the degree of Information Systems and Computer Engineering. The first column is the grade the student obtained in a given subject and the second column is the number of times the student enrolled at that subject. If the student had already been approved, we could simply fetch this information from the *subject results* star. If that was not the case, we looked for all the entries in the *subject enrollments* for that subject, and took the grade to be the grade of the latest[3] entry and the number of enrollments to be the number of records found. If we could not find any entry in the *subject enrollments*, that would mean the student never enrolled in that subject. In these cases, the grade is "NI" (short for, *não inscrito*, meaning not enrolled) and the number of enrollments is 0.

The last step was to manually annotate each of the records to classify it on whether the student quit the degree or not. To help us on this task, we added four extra columns: the number of approved subjects, the total number of enrollments, the total number of enrollments when the student reproved and finally the total number of opportunities the student had to enroll at subjects, considering that each term the student had five opportunities. Our main guide lines for the annotation were: if the student had at least 25 subjects approved then he did not quit; if he had less than 5 approved subjects having had a lot of opportunities to enroll then he quit; if the total number of enrollments is greater than or equal to the number of opportunities then he did not quit; and finally, if the last time the student enrolled at any subject was at least a year ago then he quit.

While annotating the obtained dataset, we came to the conclusion that the records relative to the 2012 freshman could not be reliably annotated. This is simple to justify: these students only had the opportunity to enroll in the subjects of two terms, or according to our opportunities column, they only had 10 opportunities to enroll. If these students have been approved at those 10 subjects, probably they have not quit, however this might not be true, as it can be noted in students in the previous years. The same is true if the student reproved the 10 subjects.

[3] The latest entry was obtained by sorting in relation to the term.

1.2 RESULT

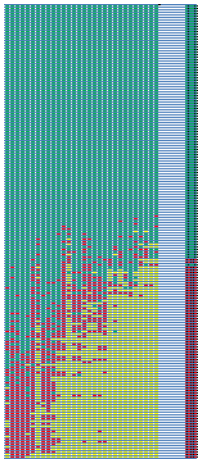


Figure 5 – Visual representation of the obtained dataset.

The final result consisted in 1181 records, of which 486 (41%) had a quit class label. Figure 5 shows a visual representation for the students of 2006. A green cell represents an approved subject, a red one a reproved subject and a yellow one a NI. The almost white column corresponds to the four extra columns and the last column is the class label, green the student did not quit, red the student quit. In this image the students were sorted on the number of approved subjects.

1.3 HOW TO GET TO BIG DATA

The resulting dataset is only in the kilobytes size range. This is clearly not big data. In order to successfully test our algorithm we replicated the dataset multiple times in order to achieve a range of different sizes: ~5 megabytes, ~50 megabytes, ~500 megabytes, ~5 gigabytes and ~50 gigabytes.

The replication was performed in two different ways:

1) By dataset: we copied the dataset *n* times and appended each copy sequentially
2) By record: we copied each record *n* times and appended each one sequentially.

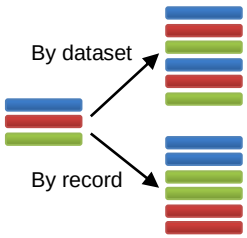


Figure 6 – Visual representation of the replication.

Since we are copying each record the same amount of times, the output of the algorithm will always be the same. We chose to replicate it in two different ways to show that the algorithm can handle unbalanced datasets and that this fact does not contribute, in a appreciable way, to its performance.

2 EXPERIMENTAL RESULTS

The experimental results were obtained by running MRID4 in the RNL cluster, with a fixed minimum support of 90%. For each dataset size and replication, the algorithm was ran 5 times. The charts we represent use the average of those 5 times.

2.1 RUN TIMES

The run time is the time it took to execute the algorithm from start to finish. This includes the computation of the translation map and all the MapReduce jobs, one for each level of the tree.

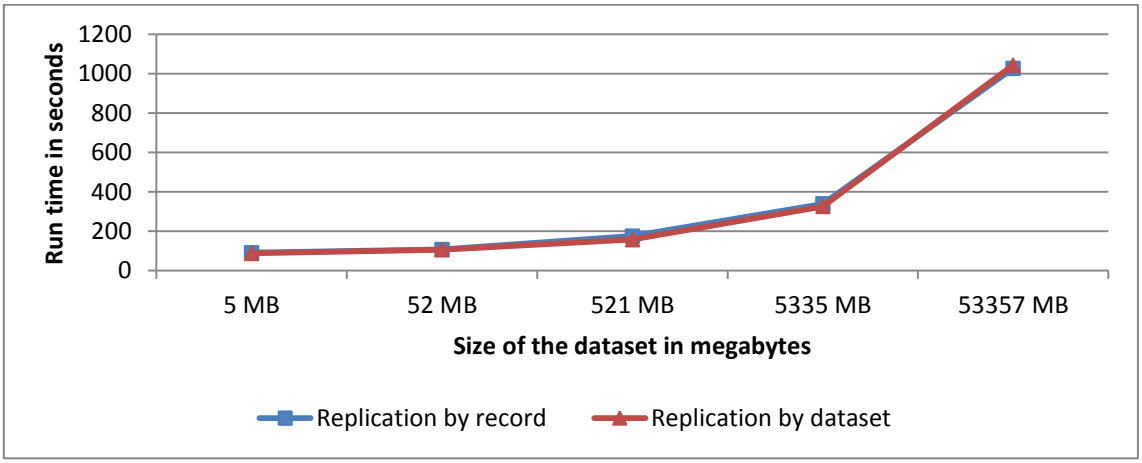


Figure 7 – Chart of the run times of the algorithm.

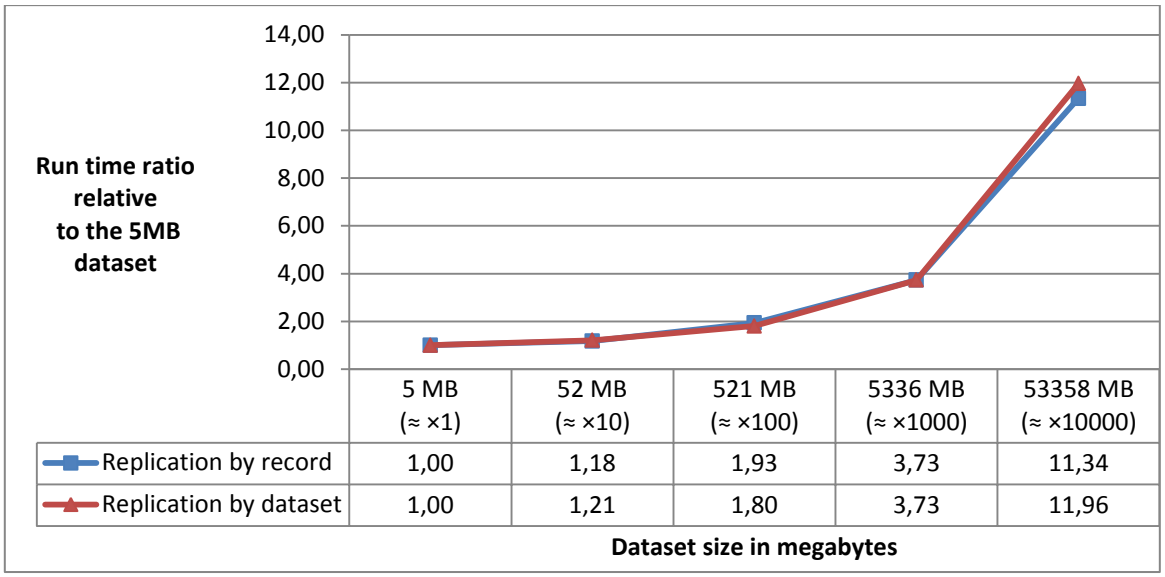
Figure 7 shows the run times of the algorithm, as the size of the dataset increases, as well as, the run times obtained with the two different replication methods.

We can see that for sizes up to ~500 megabytes the run times are almost the same. This is coherent with the general opinion that for jobs where the dataset is smaller than 1 gigabyte, Hadoop introduces more overhead than it actually helps to perform the intended task. It is also visible that the replication method had almost no impact on the run time of the algorithm.

2.2 SCALABILITY

Scalability can be measured by how much more time the system takes to process a larger data set.

Figure 8 represents the scalability of the algorithm by showing the ratio between the run time for a given dataset size and the run time for the dataset size of 5 MB. So as an example, for the dataset with 5336 MB, which is roughly 1000 times bigger than the dataset with 5 MB, the algorithm only took 3.73 times longer to run.



	5 MB (≈ ×1)	52 MB (≈ ×10)	521 MB (≈ ×100)	5336 MB (≈ ×1000)	53358 MB (≈ ×10000)
Replication by record	1,00	1,18	1,93	3,73	11,34
Replication by dataset	1,00	1,21	1,80	3,73	11,96

Figure 8 – The run time ratio relative to the 5MB dataset, as the dataset size increases.

This represents a very good scalability, which means that the algorithm with be able to run in a reasonable amount of time for very large datasets.

Chapter VIII Conclusion And Future Work

The amount of data generated by companies and entities has been growing at a very fast rate. So great in fact, that single computers can no longer handle it. And although we are surrounded by data, we are starving for information. The reason for it is simple: to extract information or meaningful knowledge from such vast amounts of data is increasingly difficult since the number of relations between the data points grows exponentially. This prompts the need to create tools that can operate at these levels of data, and that are able to do so efficiently. The MapReduce is one of such tools. It is capable of handling data sizes in the big data range, and provides the user with a simple API to do so.

In recent years there have been many applications of MapReduce to implement data mining and machine learning algorithms. Many of these implementations have had great success. However in the area of classification not much work has been done. And the focus of the work that has been done seems to be in regression trees.

In this work we introduced an easy to understand notation and approach to implement and calculate metrics such as information gain or information gain ratio and implemented a classifier, capable of building classification trees. This classifier is based on the ID3 algorithm and is built using MapReduce. We showed that although the programming model already handles many of the challenges that come with a distributed and parallel computation, there is still a need to ponder carefully on some of the aspects that arise in this environment.

We showed, through experimental results, that our algorithm able to scale to datasets with very high amounts of data.

1 Future Work

As future work we can envision some improvements to the algorithm. This is a brief explanation of each one, presenting the reasoning behind it.

1.1 Reducing The Dataset

When we explained the issues with implementing ID3 under MapReduce, we argued, in the 4th issue, that launching a job for each level of the decision tree was a very good option. The reasoning was tied to the fact that we have to traverse the decision tree for each instance, and by having a single job compute the best attribute for each node in the current level of the tree, we could ensure this computation was always useful. However, as we briefly explained in the double levels extension, as we progress in the construction of the tree we will get an increasing amount of leaf nodes. This means that not all traversals of the decision tree will be useful. This is very obvious when a large percentage, say 50%, of the instances in the dataset already falls in a leaf node for the decision tree that we have computed so far. In other words, 50% of the time spent on traversing the tree will be wasted.

A possible solution to solve this problem would be to count, in each job, how many instances fall in a leaf node. And whenever this value hits a configurable percentage of the number of instances in the dataset, we would launch a job that would simply filter out these instances in order to create a smaller dataset. From this point forward we would use this smaller dataset.

This solution comes with two caveats. First, we cannot use it when the dataset is already at a small size, say under a 1GB, because at these sizes the use of MapReduce introduces more overhead than it actually contributes. This means that even if most of the traversals of the tree are wasted, we wouldn't benefit from reducing the size of the dataset. To circumvent this caveat, in these circumstances, we could simply run the algorithm in a single computer, sequentially. This is similar to what PLANET does.

The second caveat comes from the overhead of transferring a potential big part of the dataset via the network. For example if the dataset has a size of 5TB, and we configured the aforementioned percentage to be 50% we would have to transmit about 2.5TB of data, which will undoubtedly take a considerable amount of time to do. Even if we have very high bandwidth connection between the available nodes.

1.2 When To Use Double Levels

The idea behind the double levels aims to trade computation time for storage. This tradeoff is achieved by computing two levels of the decision tree in a single job. As the code stands this option is either on or off, meaning that, either all jobs perform a double levels computation or they don't. This is problematic, since it might not be optimal in all cases.

In the first levels of the tree the algorithm can choose from almost every attribute in the dataset, this means that to implement the double levels idea each mapper needs a lot of memory. This is evident from the fact that, it not only has to maintain a Class Counts for each node of the current level of the tree, but also a Class Counts for each value, of each possible attribute to be chosen, for each node in the current level of the tree. And since we have a lot of attributes to choose from this equates to a very large amount of memory.

The solution to this problem comes in the form of a heuristic that ascertains whether activating the double levels would be beneficial. For example, if we are in the first levels of the tree, say in the first 3 levels, then it would not be beneficial to activate it.

1.3 Class Counts Implementation Regarding The Type Of Attribute

Our implementation for the class counts differs when the attribute is discrete versus when it is continuous. As it was explained, this choice was made because of the number of possible values each type of attribute normally has. Continuous attributes normally have a very high range of different attribute values, whereas discrete attributes normally have only a few different values.

However this isn’t always the case. It might be possible to have a discrete attribute with a very high range of values. In this case, the choice of using an array will not be the most appropriate, since many entries, in the mappers, will just be empty, that is, will just have zeros on all the columns.

A possible solution could be to employ the same data structure we used for continuous attributes, a map. This would lead to a more uniform implementation; however it wouldn’t be necessarily better. Another solution would be implementing a job that would estimate the distribution of the values for each attribute and use the most appropriate data structure accordingly.

Chapter IX REFERENCES

Agrawal, R., & Shafer, J. C. (1996). Parallel mining of association rules. IEEE Transactions on Knowledge and Data Engineering, 8(6), 962-969.

Agrawal, R., & Srikant, R. (1994). Fast Algorithms for Mining Association Rules in Large Databases. Proceedings of the 20th International Conference on Very Large Data Bases, 487-499.

Apache. (2013). Algorithms - Apache Mahout. Retrieved May 2013, from https://cwiki.apache.org/confluence/display/MAHOUT/Algorithms

Apache. (2013). Apache Hadoop. Retrieved May 2013, from http://hadoop.apache.org/

Apache. (2013). Apache Mahout. Retrieved May 2013, from http://mahout.apache.org/

Apache. (n.d.). PoweredBy - Hadoop Wiki. Retrieved May 2013, from http://wiki.apache.org/hadoop/PoweredBy

Breiman, L., Friedman, J., Stone, C. J., & Olshen, R. A. (1984). Classification and regression trees. Monterey, CA;: Wadsworth & Brooks/Cole Advanced Books & Software.

Buyya, R., Yeo, C. S., Venugopal, S., Broberg, J., & Brandic, I. (2009). Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Generation Computer Systems, 25(6), 599-616.

Chu, C.-T., Kim, S. K., Lin, Y.-A., Yu, Y. Y., Bradski, G., Ng, Y. A., & Olukotun, K. (2006). Map-Reduce for Machine Learning on Multicore. Advances in Neural Information Processing Systems 19, 281-288.

Condie, T., Conway, N., Alvaro, P., Hellerstein, J. M., Gerth, J., Talbot, J., . . . Sears, R. (2010). Online aggregation and continuous query support in MapReduce. Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (SIGMOD '10), 1115-1118.

Cordeiro, R. L., Junior, C. T., Traina, A. J., López, J., Kang, U., & Faloutsos, C. (2011). Clustering very large multi-dimensional datasets with MapReduce. Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '11), 690-698.

Dean, J., & Ghemawat, S. (2004). MapReduce: Simplified Data Processing on Large Clusters. OSDI'04: Sixth Symposium on Operating System Design and Implementation.

Dean, J., & Ghemawat, S. (2010). MapReduce: a flexible data processing tool. Communications of the ACM - Amir Pnueli: Ahead of His Time CACM Homepage archive, 53(1), 72-77.

Ekanayake, J., & Fox, G. (2010). High Performance Parallel Computing with Clouds and Cloud Technologies. Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, 34, 20-38.

Ene, A., Im, S., & Moseley, B. (2011). Fast clustering using MapReduce. Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '11), 681-689.

Ghemawat, S., Gobioff, H., & Leung, S.-T. (2003). The Google file system. SOSP '03 Proceedings of the nineteenth ACM symposium on Operating systems principles, 29-43.

Ghoting, A., Kambadur, P., Pednault, E., & Kannan, R. (2011). NIMBLE: a toolkit for the implementation of parallel data mining and machine learning algorithms on mapreduce. Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining, 334-342.

Google. (2013). Google Drive. Retrieved 2013, from https://drive.google.com

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I. H. (2009, June). The WEKA data mining software: an update. ACM SIGKDD Explorations Newsletter, 11(1), pp. 10-18.

Kang, U., Tsourakakis, C. E., & Faloutsos, C. (2009). PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. Proceedings of the 2009 Ninth IEEE International Conference on Data Mining, 229-238.

Lee, K.-H., Lee, Y.-J., Choi, H., Chung, Y. D., & Moon, B. (2011). Parallel data processing with MapReduce: a survey. ACM SIGMOD Record Archive, 40(4), 11-20.

Lenk, A., Klems, M., Nimis, J., Tai, S., & Sandholm, T. (2009). What's Inside the Cloud? An Architectural Map of the Cloud Landscape. CLOUD '09 Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing, 23-31.

Li, H.-G., Wu, G.-Q., Hu, X.-G., Zhang, J., Li, L., & Wu, X. (2011). K-Means Clustering with Bagging and MapReduce. Proceedings of the 2011 44th Hawaii International Conference on System Sciences (HICSS '11), 1-8.

Li, L., & Zhang, M. (2011). The Strategy of Mining Association Rule Based on Cloud Computing. Business Computing and Global Informatization (BCGIN), 475-478.

Li, N., Zeng, L., He, Q., & Shi, Z. (2012). Parallel Implementation of Apriori Algorithm Based on MapReduce. Software Engineering, Artificial Intelligence, Networking and Parallel & Distributed Computing (SNPD), 236-241.

Lin, M.-Y., Lee, P.-Y., & Hsueh, S.-C. (2012). Apriori-based frequent itemset mining algorithm on MapReduce. Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication (ICUIMC '12), (p. 8). New York.

Lv, Z., Hu, Y., Zhong, H., Wu, J., Li, B., & Zhao, H. (2010). Parallel K-means clustering of remote sensing images based on mapreduce. Proceedings of the 2010 international conference on Web information systems and mining (WISM'10), 162-170.

MacQueen, J. (1967). Some Methods for classification and Analysis of Multivariate Observations. Proceedings of Fifth Berkeley Symposium on Mathematical Statistics and Probability, 1, 281-297.

Panda, B., Herbach, J. S., Basu, S., & Bayardo, J. R. (2009). PLANET: massively parallel learning of tree ensembles with MapReduce. Proceedings of the VLDB Endowment, 2(2), 1426-1437.

Quinlan, J. R. (1986). Induction of Decision Trees. Machine Learning, 1(1), 81-106.

Quinlan, J. R. (1993). C4.5: programs for machine learning. San Francisco, CA: Morgan Kaufmann Publishers Inc.

Rao, D., & Yarowsky, D. (2009). Ranking and semi-supervised classification on large scale graphs using map-reduce. Proceedings of the 2009 Workshop on Graph-based Methods for Natural Language Processing (TextGraphs-4), (pp. 58-65). Stroudsburg.

Shvachko, K., Kuang, H., Radia, S., & Chansler, R. (2010). The Hadoop Distributed File System. MSST '10 Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies, 1-10.

Stonebraker, M., Abadi, D., DeWitt, D. J., Madden, S., Paulson, E., Pavlo, A., & Rasin, A. (2010). MapReduce and parallel DBMSs: friends or foes? Communications of the ACM - Amir Pnueli: Ahead of His Time CACM Homepage archive, 53(1), 64-71.

Vaquero, L. M., Rodero-Merino, L., Caceresand, J., & Lindner, M. (2009). A Break in the Clouds: Towards a Cloud Definition. ACM SIGCOMM Computer Communication Review, 39(1), 50-55.

Wu, G., Li, H., Hu, X., Bi, Y., Zhang, J., & Wu, X. (2009). MReC4.5: C4.5 Ensemble Classification with MapReduce. ChinaGrid Annual Conference, 4, 249-255.

Yahya, O., Hegazy, O., & Ezat, E. (2012). An Efficient Implementation of Apriori Algorithm Based on Hadoop-MapReduce Model. International Journal of Reviews in Computing, 12, 59-67.

Yin, W., Simmhan, Y., & Prasana, V. K. (2012). Scalable regression tree learning on Hadoop using OpenPlanet. Proceedings of third international workshop on MapReduce and its Applications Date (MapReduce '12), 57-64.

Zhang, C., Li, F., & Jestes, J. (2012). Efficient Parallel kNN Joins for Large Data in MapReduce. 15th International Conference on Extending Database Technology, 38-49.

Zhang, C., Li, F., & Jestes, J. (2012). Efficient Parallel kNN Joins for Large Data in MapReduce. 15th International Conference on Extending Database Technology, 38-49.

Chapter X APPENDIX

1 PROF OF THE ENTROPY FORMULA

$$\mathrm{H}(\mathbb{S}) = -\sum_{c \in \mathbb{C}} \frac{|\mathbb{S}_{y=c}|}{|\mathbb{S}|} \log_2 \frac{|\mathbb{S}_{y=c}|}{|\mathbb{S}|}$$

$$= -\frac{1}{|\mathbb{S}|} \sum_{c \in \mathbb{C}} |\mathbb{S}_{y=c}| \left(\log_2 |\mathbb{S}_{y=c}| - \log_2 |\mathbb{S}|\right)$$

$$= -\frac{1}{|\mathbb{S}|} \left(\sum_{c \in \mathbb{C}} |\mathbb{S}_{y=c}| \log_2 |\mathbb{S}_{y=c}| - \sum_{c \in \mathbb{C}} |\mathbb{S}_{y=c}| \log_2 |\mathbb{S}| \right)$$

$$= -\frac{1}{|\mathbb{S}|} \left(\frac{1}{\log 2} \sum_{c \in \mathbb{C}} |\mathbb{S}_{y=c}| \log |\mathbb{S}_{y=c}| - \frac{1}{\log 2} \sum_{c \in \mathbb{C}} |\mathbb{S}_{y=c}| \log |\mathbb{S}| \right)$$

$$= \frac{1}{|\mathbb{S}| \log 2} \left(\log |\mathbb{S}| \sum_{c \in \mathbb{C}} |\mathbb{S}_{y=c}| - \sum_{c \in \mathbb{C}} |\mathbb{S}_{y=c}| \log |\mathbb{S}_{y=c}| \right)$$

Note that the expression $\sum_{c \in \mathbb{C}} |\mathbb{S}_{y=c}|$ is equal to $|\mathbb{S}|$ because we are iterating though all the possible values for the class label.

$$= \frac{1}{|\mathbb{S}| \log 2} \left(|\mathbb{S}| \log |\mathbb{S}| - \sum_{c \in \mathbb{C}} |\mathbb{S}_{y=c}| \log |\mathbb{S}_{y=c}| \right)$$

$$= \frac{|\mathbb{S}| \log |\mathbb{S}| - \sum_{c \in \mathbb{C}} |\mathbb{S}_{y=c}| \log |\mathbb{S}_{y=c}|}{|\mathbb{S}| \log 2}$$

2 COMPUTING THE INFORMATION GAIN RATIO FROM THE CLASS COUNTS

The information gain ratio is defined as the ratio between the information gain and the intrinsic value.

$$\mathrm{IGR}(\mathbb{S}, \mathrm{a}) = \frac{\mathrm{IG}(\mathbb{S}, \mathrm{a})}{\mathrm{IV}(\mathbb{S}, \mathrm{a})}$$

Where the intrinsic value is defined as:

$$\mathrm{IV}(\mathbb{S}, \mathrm{a}) = -\sum_{v \in \mathrm{vals}(a)} \frac{|\mathbb{S}_{x_a=v}|}{|\mathbb{S}|} \log_2 \frac{|\mathbb{S}_{x_a=v}|}{|\mathbb{S}|}$$

Following the steps presented in point 1 we can derive the intrinsic value to be:

$$\mathrm{IV}(\mathbb{S}, \mathrm{a}) = \frac{|\mathbb{S}| \log|\mathbb{S}| - \sum_{\mathrm{v} \in \mathrm{vals(a)}} \left|\mathbb{S}_{\mathrm{x_a=v}}\right| \log\left|\mathbb{S}_{\mathrm{x_a=v}}\right|}{|\mathbb{S}| \log 2}$$

The major difference from what was presented in point 1 is that instead of $\sum_{\mathrm{c} \in \mathbb{C}} \left|\mathbb{S}_{\mathrm{y=c}}\right|$, we will now have $\sum_{\mathrm{v} \in \mathrm{vals(a)}} \left|\mathbb{S}_{\mathrm{x_a=v}}\right|$, however this will still be equal to $|\mathbb{S}|$ since we are iterating though all the possible values of the attribute a.

Considering the formulas for the entropy and the information gain:

$$\mathrm{H}(\mathbb{S}) = \frac{|\mathbb{S}| \log|\mathbb{S}| - \sum_{\mathrm{c} \in \mathbb{C}} \left|\mathbb{S}_{\mathrm{y=c}}\right| \log\left|\mathbb{S}_{\mathrm{y=c}}\right|}{|\mathbb{S}| \log 2}$$

$$\mathrm{IG}(\mathbb{S}, \mathrm{a}) = \mathrm{H}(\mathbb{S}) - \frac{\sum_{\mathrm{v} \in \mathrm{vals(a)}} \left(\left|\mathbb{S}_{\mathrm{x_a=v}}\right| \log\left|\mathbb{S}_{\mathrm{x_a=v}}\right| - \sum_{\mathrm{c} \in \mathbb{C}} \left|\mathbb{S}_{\mathrm{x_a=v} \wedge \mathrm{y=c}}\right| \log\left|\mathbb{S}_{\mathrm{x_a=v} \wedge \mathrm{y=c}}\right|\right)}{|\mathbb{S}| \log 2}$$

And introducing the function $nlogn(x) = \mathrm{x} * \log(\mathrm{x})$

The information gain ratio is:

$$\mathrm{IGR}(\mathbb{S}, \mathrm{a}) = \frac{\frac{|\mathbb{S}| \log|\mathbb{S}| - \sum_{\mathrm{c} \in \mathbb{C}} nlogn\left(\left|\mathbb{S}_{\mathrm{y=c}}\right|\right)}{|\mathbb{S}| \log 2} - \frac{\sum_{\mathrm{v} \in \mathrm{vals(a)}} \left[nlogn\left(\left|\mathbb{S}_{\mathrm{x_a=v}}\right|\right) - \sum_{\mathrm{c} \in \mathbb{C}} nlogn\left(\left|\mathbb{S}_{\mathrm{x_a=v} \wedge \mathrm{y=c}}\right|\right)\right]}{|\mathbb{S}| \log 2}}{\frac{|\mathbb{S}| \log|\mathbb{S}| - \sum_{\mathrm{v} \in \mathrm{vals(a)}} nlogn\left(\left|\mathbb{S}_{\mathrm{x_a=v}}\right|\right)}{|\mathbb{S}| \log 2}}$$

$$= \frac{|\mathbb{S}| \log|\mathbb{S}| - \sum_{\mathrm{c} \in \mathbb{C}} nlogn\left(\left|\mathbb{S}_{\mathrm{y=c}}\right|\right) - \sum_{\mathrm{v} \in \mathrm{vals(a)}} \left[nlogn\left(\left|\mathbb{S}_{\mathrm{x_a=v}}\right|\right) - \sum_{\mathrm{c} \in \mathbb{C}} nlogn\left(\left|\mathbb{S}_{\mathrm{x_a=v} \wedge \mathrm{y=c}}\right|\right)\right]}{|\mathbb{S}| \log|\mathbb{S}| - \sum_{\mathrm{v} \in \mathrm{vals(a)}} nlogn\left(\left|\mathbb{S}_{\mathrm{x_a=v}}\right|\right)}$$

$$= \frac{|\mathbb{S}| \log|\mathbb{S}| - \sum_{\mathrm{c} \in \mathbb{C}} nlogn\left(\left|\mathbb{S}_{\mathrm{y=c}}\right|\right) - \sum_{\mathrm{v} \in \mathrm{vals(a)}} nlogn\left(\left|\mathbb{S}_{\mathrm{x_a=v}}\right|\right) - \sum_{\mathrm{v} \in \mathrm{vals(a)}} \sum_{\mathrm{c} \in \mathbb{C}} nlogn\left(\left|\mathbb{S}_{\mathrm{x_a=v} \wedge \mathrm{y=c}}\right|\right)}{|\mathbb{S}| \log|\mathbb{S}| - \sum_{\mathrm{v} \in \mathrm{vals(a)}} nlogn\left(\left|\mathbb{S}_{\mathrm{x_a=v}}\right|\right)}$$

$$= \frac{|\mathbb{S}| \log|\mathbb{S}| - \sum_{\mathrm{v} \in \mathrm{vals(a)}} nlogn\left(\left|\mathbb{S}_{\mathrm{x_a=v}}\right|\right) - \sum_{\mathrm{c} \in \mathbb{C}} nlogn\left(\left|\mathbb{S}_{\mathrm{y=c}}\right|\right) - \sum_{\mathrm{v} \in \mathrm{vals(a)}} \sum_{\mathrm{c} \in \mathbb{C}} nlogn\left(\left|\mathbb{S}_{\mathrm{x_a=v} \wedge \mathrm{y=c}}\right|\right)}{|\mathbb{S}| \log|\mathbb{S}| - \sum_{\mathrm{v} \in \mathrm{vals(a)}} nlogn\left(\left|\mathbb{S}_{\mathrm{x_a=v}}\right|\right)}$$

$$= \frac{|\mathbb{S}| \log|\mathbb{S}| - \sum_{\mathrm{v} \in \mathrm{vals(a)}} nlogn\left(\left|\mathbb{S}_{\mathrm{x_a=v}}\right|\right) - \sum_{\mathrm{c} \in \mathbb{C}} \left[nlogn\left(\left|\mathbb{S}_{\mathrm{y=c}}\right|\right) - \sum_{\mathrm{v} \in \mathrm{vals(a)}} nlogn\left(\left|\mathbb{S}_{\mathrm{x_a=v} \wedge \mathrm{y=c}}\right|\right)\right]}{|\mathbb{S}| \log|\mathbb{S}| - \sum_{\mathrm{v} \in \mathrm{vals(a)}} nlogn\left(\left|\mathbb{S}_{\mathrm{x_a=v}}\right|\right)}$$

$$= 1 - \frac{\sum_{\mathrm{c} \in \mathbb{C}} \left[nlogn\left(\left|\mathbb{S}_{\mathrm{x_a=v}}\right|\right) - \sum_{\mathrm{v} \in \mathrm{vals(a)}} nlogn\left(\left|\mathbb{S}_{\mathrm{x_a=v} \wedge \mathrm{y=c}}\right|\right)\right]}{|\mathbb{S}| \log|\mathbb{S}| - \sum_{\mathrm{v} \in \mathrm{vals(a)}} nlogn\left(\left|\mathbb{S}_{\mathrm{x_a=v}}\right|\right)}$$

$$= 1 - \frac{\sum_{\mathrm{c} \in \mathbb{C}} \left[\left|\mathbb{S}_{\mathrm{y=c}}\right| \log\left|\mathbb{S}_{\mathrm{y=c}}\right| - \sum_{\mathrm{v} \in \mathrm{vals(a)}} \left|\mathbb{S}_{\mathrm{x_a=v} \wedge \mathrm{y=c}}\right| \log\left|\mathbb{S}_{\mathrm{x_a=v} \wedge \mathrm{y=c}}\right|\right]}{|\mathbb{S}| \log|\mathbb{S}| - \sum_{\mathrm{v} \in \mathrm{vals(a)}} \left|\mathbb{S}_{\mathrm{x_a=v}}\right| \log\left|\mathbb{S}_{\mathrm{x_a=v}}\right|}$$

And just like for the information gain, if we have $\left|\mathbb{S}_{x_a=v \wedge y=c}\right|$ for each a, v and c (the class counts) we can compute the information gain ratio.