POLITECNICO DI MILANO
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

Scaling feature selection algorithms using MapReduce on Apache Hadoop

AI & R Lab
Artificial Intelligence and
Robotics Laboratory
of Politecnico di Milano

MLG
Machine Learning Group
Université Libre de Bruxelles

Supervisor: Prof. Andrea Bonarini
Co-supervisor: Prof. Gianluca Bontempi

Candidate:
Claudio Reggiani, 771198

Academic Year 2012-2013

To my parents, who always supported and pushed me into pursuing my academic and life goals.

Abstract

Domains such as internet algorithms, computational biology or social link analysis deal with very large datasets, also called Big Data. In these scenarios, single machine learning algorithms cannot handle easily the entire data to produce models and MapReduce emerged as new paradigm to query data in distributed systems. Apache Hadoop is the result of the open source community in letting this technology available for everyone.

There is a huge amount of information locked up in databases and these information can be exploited with machine learning techniques, which aim to find and describe structural patterns in data. Our work focused on feature selection algorithms because they generalize prediction models reducing the noise in input. We implemented two state-of-the-art feature selection algorithms (Ranking and mRMR) according to the MapReduce programming paradigm and we made them available to the open source Big Data community.

In order to visualize and model the scalability of the approach, we ran several and extensive performance tests at the university computing center, where we first built a user friendly service which dynamically deploys Apache Hadoop in the cluster. The results mostly matched our expectations and theoretical analysis, when algorithms ran over large datasets.

Extended abstract

In questa tesi esaminiamo quali sono le limitazioni dei tradizionali Database Management Systems nel gestire grandi collezioni di dati. Questi sono stati in passato la soluzione di riferimento per salvare in modo organizzato i dati, tuttavia nei sistemi distribuiti non rappresentano la migliore soluzione, a causa della complessità e dei vincoli introdotti. Con l'avvento del Big Data è stato necessario un nuovo paradigma per immagazzinare e interrogare i dati per superare i limiti di tali architetture. Le nuove applicazioni emerse sono state raggruppate sotto il termine NoSQL.

Nel 2003 e 2004, Google ha pubblicato due articoli, Google file system e MapReduce, che sono le fondamenta del nuovo modo di immagazzinare e interrogare i dati. Apache Hadoop è il risultato della communità open source nel rendere tale tecnologia disponibile a tutti e un intero nuovo ecosistema di software proprietari e open è nato con lo scopo di arricchirne le funzionalità.

Noi eravamo interessati nello studiare e implementare algoritmi di machine learning in MapReduce nel contesto del Big Data. Si parla di Big Data per esempio in biologia computazionale, analisi di social network e text mining, dove gli algoritmi tradizionali eseguibili localmente su singole macchine non riescono a produrre modelli in tempi ragionevoli. Abbiamo posto la nostra attenzione sugli algoritmi di feature selection per due motivi: perché sono tecniche per ridurre il rumore e informazioni irrilevanti dal dataset, permettendo una migliore generalizzazione del modello di predizione e perché non sono disponibili nei software open source, come Apache Mahout.

Abbiamo implementato gli algoritmi di mRMR e Ranking in MapReduce come una libreria Java integrabile ad Apache Mahout e rilasciato il codice open source su GitHub. Il nostro scopo è stato quello di studiare la scalabilità di tali algoritmi e considerare quanto l'overhead di Apache Hadoop, nel gestire il cluster, incide nel tempo totale di esecuzione. Nonostante le ridotte risorse a disposizione, in relazione ad altre ricerche, abbiamo potuto fare le seguenti considerazioni: per piccoli dataset o nel caso in cui i parametri

del cluster siano male impostati, l'overhead condiziona in modo significativo il tempo finale. Con grandi dataset i risultati sono stati coerenti con le nostre aspettative, ovvero che il tempo di esecuzione è proporzionale alla dimensione dell'input e inversamente proporzionale alla quantità di risorse computazionali disponibili, i.e al numero di nodi e core nel cluster.

I test sono stati condotti utilizzando il centro di calcolo (HPC) universitario, nel quale abbiamo esplorato le possibili alternative per integrare Apache Hadoop nel cluster. Abbiamo deciso di sviluppare un servizio sul modello di Amazon Elastic MapReduce, user friendly e disponibile per qualsiasi utente. Tale servizio, in modo automatico e dinamico, installa e configura Apache Hadoop sui nodi ed esegue il job sottomesso. Gli unici parametri che l'utente deve impostare sono il numero di nodi del cluster e definire quale job eseguire.

Acknowledgements

Many people contributed to this experience. First of all, I am very grateful to my mother and father for supporting me during this journey since the beginning, even when the earthquakes that struck in May 2012 heavily affected the village we live in and our lifes.

I want to thank Professor Andrea Bonarini for giving me the opportunity to spend the master thesis abroad and Professor Gianluca Bontempi for hosting me in the Machine Learning Group (MLG)[1] at Université Libre de Bruxelles (ULB). During the year they steadily followed my research and gave me important feedbacks. Furthermore I got the opportunity to present my research to enterprises and to attend local and international conferences. Among MLG members a special thank goes to Yann-Aël Le Borgne and Andrea Dal Pozzolo for advices and guidance.

This thesis is the last chapter of a great adventure in Politecnico di Milano, which would not have been the same without all the volleyball players and coursemates I met during the master program, especially Fabio Forghieri with whom I started the journey.

This research project won Spinner[2] grant for innovative projects and ideas devoted to people in collaboration with universities, research centers and enterprises in Emilia Romagna. This achievement has been possible with the collaboration of AIRLab[3], MLG, DataRiver[4] and ATOS Worldline[5].

[1]http://mlg.ulb.ac.be/
[2]http://www.spinner.it/
[3]http://airlab.ws.dei.polimi.it
[4]http://www.datariver.it/
[5]http://atosworldline.be

Contents

Abstract **I**

Extended abstract **III**

Acknowledgements **VII**

1 Introduction **1**
- 1.1 Big Data and NoSQL . . . 1
- 1.2 Technology overview . . . 2
- 1.3 Machine Learning . . . 5
- 1.4 Work description and contributions . . . 7
- 1.5 Thesis Structure . . . 8

2 State of the art **9**
- 2.1 Database management systems . . . 9
  - 2.1.1 ACID Properties . . . 9
  - 2.1.2 Why sharding a database . . . 10
  - 2.1.3 How sharding a database . . . 10
  - 2.1.4 Problems in sharding . . . 11
  - 2.1.5 CAP Theorem . . . 12
  - 2.1.6 Changing paradigm . . . 12
- 2.2 Apache Hadoop . . . 13
  - 2.2.1 HDFS Overview . . . 14
  - 2.2.2 HDFS architecture and Data Replication . . . 14
  - 2.2.3 Robustness . . . 16
  - 2.2.4 Web interface . . . 16
- 2.3 MapReduce . . . 17
  - 2.3.1 Parallel paradigms . . . 17
  - 2.3.2 Data type . . . 18
  - 2.3.3 Core components . . . 18
  - 2.3.4 Word count example . . . 19

2.3.5 MapReduce data flow 20
2.3.6 Combiner 22
2.3.7 Distributed cache 22
2.3.8 Reading and writing 23
2.3.9 MapReduce architecture in Apache Hadoop 25
2.3.10 Language support 26
2.4 Analytics with Big Data 27
2.4.1 Apache HBase 27
2.4.2 Apache Hive 28
2.4.3 MLbase 29
2.4.4 R programming language 30
2.5 Apache Mahout 31
2.5.1 Apache Mahout and Apache Hadoop 31
2.5.2 Tools and algorithms 31
2.5.3 Alternatives to Apache Mahout 34

3 Feature Selection algorithms in MapReduce 35
3.1 Introduction to feature selection 36
3.1.1 Dependence and correlation 37
3.1.2 Mutual Information 37
3.1.3 Algorithm categories 38
3.1.4 Minimal Redundancy Maximal Relevance 40
3.2 Feature selection in MapReduce 41
3.3 Contributions 42
3.3.1 Mutual information Implementation 43
3.3.2 Ranking implementation 43
3.3.3 mRMR implementation 45

4 Deployment 49
4.1 Amazon Elastic MapReduce 49
4.2 Hydra, the computing center 51
4.2.1 Hydra as batch system 52
4.2.2 User jobs in Hydra 53
4.2.3 Torque and Moab 54
4.2.4 Deploying Apache Hadoop on Hydra 56
4.3 Contributions 57
4.3.1 Job submission 58
4.3.2 Automatic deployment 58
4.3.3 Considerations 59
4.3.4 How to use the service 59

5 Performance tests and results **63**
5.1 Datasets . . . 63
5.2 Tests and considerations . . . 65
5.2.1 Apache Hadoop setup . . . 66
5.2.2 Retrieving information . . . 66
5.3 mRMR MapReduce job . . . 67
5.3.1 Parallelism tests . . . 67
5.3.2 Overhead tests . . . 67
5.3.3 Scalability tests . . . 70
5.4 Best feature MapReduce job . . . 72
5.5 Ranking MapReduce job . . . 75
5.5.1 Parallelism tests . . . 75
5.5.2 Scalability tests . . . 75

6 Future works and conclusions **79**

Glossary **81**

List of variables **83**

Bibliography **85**

A Apache Mahout tools **91**
A.1 Preprocessing . . . 91
A.2 Algorithms . . . 92
A.3 Postprocessing . . . 94
A.4 Utilities . . . 94

B Mutual Information for MapReduce **97**

C mRMR in MapReduce **101**

D Ranking in MapReduce **105**

E Tests **107**

Chapter 1

Introduction

"I was not thinking about challenging database technologies at all"
Doug Cutting, Apache Hadoop founder

1.1 Big Data and NoSQL

With the cost of a gigabyte decreased from $10 in 2000 down to $0.10 in 2010, more and more companies are investing in commodity hardware to store an increasing amount of data over time. In 2012, data accounted for more than 30 billion pieces of content added each month on Facebook, 1 petabyte of content processed everyday by Zynga, 2 billion videos watched everyday on YouTube, 32 billion searches performed each month on Twitter.

The traditional database systems, such as relational databases, have been pushed to the limit and when data exceeds their processing capacity these systems are facing *Big Data* issues. There is no official definition of Big Data and we refer to three terms that are commonly used to characterize it [35]:

- *Volume*: volume presents the most immediate challenge to conventional structures. It calls for scalable storage, and a distributed approach to querying. Many companies have already large amounts of archived data, perhaps in the form of logs, but not the capacity to process it.
- *Velocity*: it is the rate at which data flows into an organization. It includes both the inflow data (the input from different sources) and outflow data (batch processing).
- *Variety*: Big Data is any type of data, it can be structured, semistructured and unstructured data such as text, sensor data, audio, video, click streams, log files and more.

To tackle the challenges of Big Data, a new breed of technologies has emerged. Many of these new technologies have been grouped under the term *NoSQL*, which stands for "Not Only SQL" and it groups a subset of storage software that is designed with the intention of increasing optimization for high-performance operations on large dataset. These technologies provide a mechanism for storage and retrieval of data in order to achieve horizontal scaling (see Glossary) and higher availability. Graphs, time series and genomics data cannot be modeled well in traditional database systems and different approaches and solutions emerged. It is worth noticing that differences between NoSQL databases are bigger than between SQL ones and the best solution depends specifically on the project properties and goals.

NoSQL technology allows us to capture and store vast quantities of data. If data are recorded facts, then information is the set of patterns that underlies the data. There is a huge amount of information locked up in databases and this information can be exploited with machine learning techniques, which aim to find and describe structural patterns in data [61]. The kind of descriptions found can be used for prediction, explanation, and understanding. Some applications focus on prediction: forecasting what will happen in new situations from data that describes what happened in the past. When the new database system replaces the traditional one the algorithms have to be reshaped to fit the new system.

The goal of the thesis is to understand what is the state of the art technologies that managed Big Data and in which ways it is possible to make use of machine learning algorithms on such distributed systems. Both closed and open source solutions are available and the whole ecosystem is growing over time because an increasing amount of companies are shifting from the traditional model to the new one. The main research contribution of the thesis is the design and implemention of state-of-the-art machine learning (specifically feature selection) algorithms in the Big Data framework.

1.2 Technology overview

Data processing problems can be grouped in two main categories: *computation intensive* and *data intensive*. In computation intensive problems, such as π calculation, computational resource demands are high even though the amount of data loaded is low and a mainframe retrieves all data from different sources in the main memory for processing. The mainframe has usually high computational and system resources in order to run jobs efficiently. On the other hand, data intensive problems apply few calculations on data. When it comes to Big Data, it is not feasible anymore to move and load all

Google project	**Open Source project**
Google File System MapReduce	Apache Hadoop
BigTable	Apache HBase
Pregel	Apache Giraph
Dremel	Apache Drill

Table 1.1: Google projects and their counterparts open source

data in the main memory, the solution is to move computation where data resides. This latter paradigm affects the way algorithms are implemented and they have often to be reshaped to fit the new system.

Since 2003, Google have published researches in large distributed data-intensive applications and its contributions led the development of open-source technologies. Table 1.1 sums up Google projects and their open source counterparts. The research papers regarding *Google File System* [21] and *MapReduce* [14], published in 2003 and 2004 respectively, are the cornerstone to tackle Big data problems, as they provide a scalable way to store and query data and they were followed by high level projects such as Bigtable [8] and Pregel [42]. On the open source side, a whole ecosystem is growing around *Apache Hadoop*[1] and it provides both advanced and high-level tools to approach data intensive problems with Big Data. Some examples are database-like solutions, system monitoring or high-level language for data analysis. Figure 1.1 is an overview of some tools developed on top of Apache Hadoop.

Companies can leverage Apache Hadoop ecosystem in three different ways: getting full expertise in deploying Apache Hadoop and building data analytics tools, relying on external cloud infrastructures such as Amazon Elastic MapReduce (EMR)[3] which provides an optimized Apache Hadoop cluster, or getting full support from companies like Cloudera[4], Hortonworks[5] and Greenplum[6] which provide ad-hoc all-in-one platform to meet enterprise needs.

The Big Data landscape is broader than the sole Apache Hadoop ecosystem. Since NoSQL covers a wide range of technologies, one way to distin-

[1]http://hadoop.apache.org/
[2]http://www.stanford.edu/class/ee380/Abstracts/111116-slides.pdf
[3]http://aws.amazon.com/elasticmapreduce/
[4]http://www.cloudera.com
[5]http://www.hortonworks.com/
[6]http://www.greenplum.com

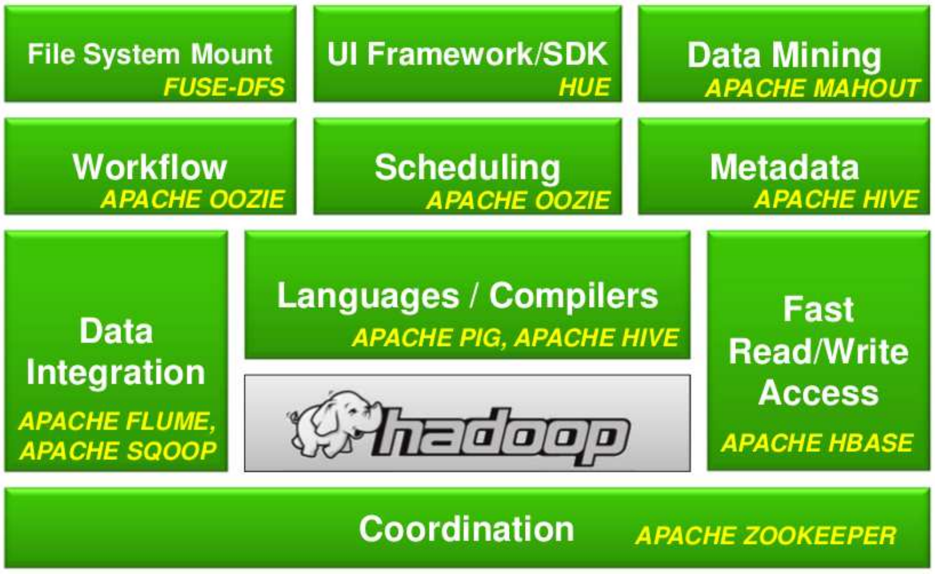


Figure 1.1: Apache Hadoop ecosystem[2]

guish NoSQL applications is by understanding what type of data structure they store. These categories are described in the following list:

1. *Key-values stores*: data is stored and indexed based on its key. This model is the simplest one, but it is inefficient with high complexity data. Examples are Voldemort[7] and Riak[8].

2. *Column Family Stores*: data is stored column-wise instead of record-wise for better compression. Here keys point to multiple columns, arranged by column family. Examples are Apache HBase (Section 2.4.1) and Apache Cassandra[9]. Column oriented Relational DBMS, such as C-Store [56], apply the same concept but data is still organized in tables.

3. *Document Databases*: they are essentially the next level of key-values, allowing nested values associated with each key. Examples are CouchDB[10] and MongoDB[11]. In contrast to RDBMS and their notions of relations (or tables), these systems are designed around an abstract notion of a *Document*.

[7]http://www.project-voldemort.com
[8]http://basho.com/riak/
[9]http://cassandra.apache.org/
[10]http://couchdb.apache.org/
[11]http://www.mongodb.org/

4. *Graph Databases*: a flexible graph model is used instead of tables of rows and columns and the rigid structure of SQL. Examples are Neo4j[12] and FlockDB[13]. In graph databases, the relationships are stored at the individual record level, while in a RDBMS they are defined at a higher level (the table definitions).

In NoSQL applications schema (see Glossary) can be defined at read-time and this flexibility is a crucial property to deal with semi and unstructured data, while in DBMS schema must be defined before any data is loaded.

Besides the hype related to Big Data, there is no one-over-all winner between DBMS and NoSQL technologies. According to its properties, each specific problem best fits with one solution or the other[14].

1.3 Machine Learning

Machine learning is a branch of Computer Science which studies how systems learn tasks. The definition of learning in this context is quite broad and we consider the one provided by Tom Mitchell in his book "Machine Learning" [46]:

> *"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."*

There are three general classes of tasks in machine learning: *supervised learning* wants to predict an output when given an input, *reinforcement learning* learns to select an action to maximize payoff and *unsupervised learning* discovers a good internal representation of the input.

In each class we want to find a model that approximates the real underlying structure of the data, so that it will also approximate the target function well over other unobserved examples. We measure the model performance when it runs over both training and test dataset. If the model performs well with the former but not with the latter it *overfits* data, which means it explains the training dataset too accurately but it is not general enough to be accurate for previously unseen data.

Overfitting occurs when a model mistakenly fits noise along with the signal. There are several approaches to prevent it such as cross-validation and

[12]http://www.neo4j.org/

[13]https://github.com/twitter/flockdb

[14]http://www.stanford.edu/class/ee380/Abstracts/111116-slides.pdf

regularization terms. In the former the whole dataset is splitted into training set, which is used to build the model, and validation set, for parameter optimization; in the latter some coefficients are used to penalize complexity. In this document, we focus on how to get rid of irrelevant features in order to reduce the variance of the model and the overfitting risk.

Curse of dimensionality

In applied mathematics, the curse of dimensionality refers to the problem caused by the increasing in volume associated with adding extra dimensions to a mathematical space. It might be difficult to work with high dimensional data since adding new features can increase the noise and it requires more observation to get good estimates. In the Big Data scenario it is easy to store as much data as possible, even lots of features.

In predicting the desired output, some features only might be relevant, others might be completely irrelevant or correlated. Performance of decision tree algorithms (such as ID3, C4.5, and CART) and instance-based algorithms (such as IBL) are affected by irrelevance, while Naive-Bayes performance is affected by redundant features [31].

Dimensionality reduction

One way to address the curse of dimensionality is by means of dimensionality reduction techniques. There are several benefits in projecting the original data into a lower dimensional space, such as measurement and computation costs, model generalization, interpretation of the model, visualization of data and understanding key features of the data.

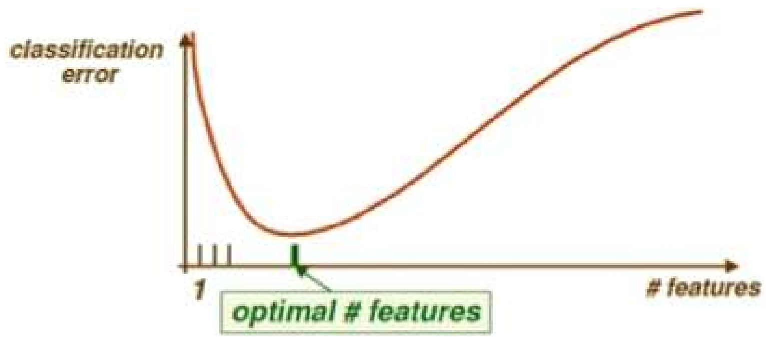


Figure 1.2: Classification error as a function of the number of feature selected[15].

[15]http://www.slideshare.net/NikhilSharma6/curse-of-dimensionality

The ultimate goal of those techniques is to provide the optimal feature set through which the data is best expressed. Figure 1.2 shows the classification error, which arises when the number of features is either less or more than the optimal one. In the first case we are missing important features that express data, while in the second case some inputs are superfluous.

Algorithms for dimensionality reduction are grouped in two main categories:

- *Feature selection*: it selects a subset of the existing features without a transformation. Examples are mRMR [51] and CFS [27] (see Section 3.1.3 for details).
- *Feature extraction*: it transforms the existing features into a lower dimensional space. Examples are PCA [28] and ICA [30].

In Chapter 3 we describe feature selection only, since the algorithms we implemented in MapReduce belong to that category.

1.4 Work description and contributions

The master thesis work is divided into two main parts. First, we introduce the context of the work, exploring Big Data framework and higher-level tools. Second, we describe our contributions both for Big Data and machine learning, which are summed up in the following list (more details at the relative chapters):

- Analyze and design filter feature selection algorithms in MapReduce. Release their implementation as a public Java library available for Apache Mahout on GitHub[16] (Chapter 3).
- Develop a user friendly service for dynamic deploying of Apache Hadoop for the ULB high-performance computing center, based on a simpler model of Amazon Elastic MapReduce[17] (Chapter 4).
- Run performance tests of implemented algorithms in MapReduce for assessing their scalability and level of parallelism (Chapter 5).

We created several datasets to understand Apache Hadoop scaling behaviour and we ran massive and extensive performance tests on clusters of different size: 4, 6 and 8 nodes. Our theoretical analysis confirmed that

[16]https://github.com/Nophiq/Mahout

[17]http://aws.amazon.com/elasticmapreduce/

execution time increases as the dataset grows in size, while it decreases as the cluster has more computational resources, i.e. nodes. Our results agreed with our expectations.

We faced several challenges during our research. We were first in developing the Apache Hadoop service on the ULB computing center, so we could not rely on any documentation and we got the expertise on the way. We needed to study and reshape machine learning algorithms to fit MapReduce paradigm, with accurate designing choices. Finally, even though our extensive performance tests agreed with our expectations, we were heavily limited by computational resources available. On the basis of our current results some future perspectives are opened:

- There are several feature selection algorithms besides the ones implemented. The Apache Mahout extension library could be enriched with new algorithms such as those listed in [47].
- There are several ways to tune Apache Hadoop or the cluster. Setting different parameters will enrich the comprehension of the algorithm workloads. The final goal then will be to build a performance model in order to predict what is the best Apache Hadoop configuration and cluster size based on the algorithm and the dataset.
- Additional extensive tests can be run over clusters with more computational resources.
- Reimplement our MapReduce algorithms with different design choices. Run new tests and compare the results.

1.5 Thesis Structure

The thesis is structured as follows. In Chapter 2 we will introduce the state of the art of technolgies: MapReduce paradigm, Apache Hadoop, Apache Mahout, database-like solutions for Big Data and how R programming language can be combined with Apache Hadoop. In later chapters, we describe limitations of the current state of the art, what motivated us in going through this master thesis work and our contributions: Chapter 3 introduce feature selection and our implemented algorithms: mRMR and Ranking; in Chapter 4 we describe the ULB computing center architecture, its limitations for Apache Hadoop and how we developed an user friendly service available for every user; in Chapter 5 we show and comment performance results. Finally, we provide our conclusions and future works in Chapter 6.

Chapter 2

State of the art

"The biggest hurdle was training our brains to think MapReduce"

Alex Holmes, author of *Hadoop in Practice*

In this chapter we describe the technology used during the master thesis. We start with a review of database management systems (DBMS), introducing the properties that made them successful and what are their limits. In distributed systems, they were forced to fit new constraints and some trade-offs were made. From one side the expertise gained over the years was still relevant, but on the other side the complexity of the system increased.

We review NoSQL technology and the scienfitic research that led the development of MapReduce, a new paradigm built to overcome scalability issue of DBMS, implemented in the open source project Apache Hadoop. A whole new ecosystem of closed and open sourced software is built on top of Apache Hadoop for different goals: analytics, visualization, management, monitoring, etc... For the purpose of the master thesis we focused on Apache Mahout, as analytics tool.

2.1 Database management systems

In these section we describe database management systems (DBMS) with its ACID properties and how they increased in complexity to deal with scalability. The aim is not to provide a fully description of DBMS but to emphasize the limits which allowed NoSQL applications to emerge.

2.1.1 ACID Properties

A database management system (DBMS) is a software system to manage, define and query databases. Each operation that modifies the database or its content is handled by the DBMS in a reliable way by means of database transactions, the execution of which must satisfy some properties referred with the acronym ACID [26]. The ACID properties are described as follow:

- *Atomicity*: whenever a database transaction starts either all or none operations involved are executed successfully. If one operation fails before the transaction ends, the valid system status before that transaction began has to be restored.
- *Consistency*: each transaction brings from one database valid state to another one, where each defined constraint is satisfied.
- *Isolation*: the DBMS manages transactions as if they were executed sequentially and it is in charge of managing concurrent transactions.
- *Durability*: once a transaction has been committed its effects are stored in such a way they will permanently stay.

2.1.2 Why sharding a database

When the DBMS wants to both serve an increasing amount of clients and guarantee ACID properties, the service response time is not reasonable and the computational resources are no more sufficient. At this point, it is time to scale the architecture. There are two approaches: *scaling-up* increases the disk storage, memory capacity, I/O performance and the other general computational resources of a single controller for processing, while *scaling-out* distributes performance and capacity among several controllers.

The scale-up approach offers simple and central configuration of the main architecture, but as data grows its performance is limited by the capabilities of its components, whose price increases as their requirements become higher. If the service has lots of operations regarding the storage system and the main memory, then the primary bottleneck will likely be I/O. In such cases, managing a single server in charge of all the workload is not affordable and a different approach has to be embraced. The scale-out approach provides a cost-saving solution by sharding (see Glossary) the database at the cost of increasing the complexity of the architecture, and it is the solution usually adopted.

2.1.3 How sharding a database

There are different ways to break up a single database into multiple ones. They could be grouped in three broad categories[1]:

[1]http://www.startuplessonslearned.com/2009/01/sharding-for-startups.html

- *Vertical partitioning*: data related to a specific feature of the product is stored in the same machine. For instance, partitioning Twitter data could be done in the following way: user information is stored in the first server, user posts in the second one and followers in a third one. Note that each feature could have different tables, in the Twitter example both tweets and direct messages belong to the same feature.

- *Key or hash based partitioning*: data is partitioned based on a key (or hash value), which is used at query time to determine where the entity is in the cluster of servers. In the Twitter example, the user unique identifier is the input of an hash function, which output shows in which server the user information is stored.

- *Directory-based partitioning*: information about how the data is shared across the cluster in store in a lookup table. There are two main drawbacks, first the machine that stores the lookup table becomes the single point of failure, second there is a performance cost since each query consults the lookup table.

2.1.4 Problems in sharding

Sharding introduces complexity in the architecture in terms of new constraints. These constraints are related to operations that involve multiple tables or instances that are not in the same server. Some of these constraints are here described.

A *distributed join* is the most expensive distributed data analysis operation. Once a database is sharded across multiple servers, the join operation has to query all databases in which the tables or instances are stored. This is an inefficient solution, since it could dramatically decrease performance of the system. A workaround is to *denormalize* (see Glossary) the database, duplicating data across servers in order to reduce join operations. This introduce even more complexity in the system because data inconsistency here has to be taken into account.

Integrity controls become more complicated in a sharded system due to distribution. We need to define where integrity rules are stored and which tables are affected or referenced by those rules. The introduced overhead is proportional to the number of shards and data replication [52]. Even though performance improvements have been achieved [18], enforcing data integrity constraints such as foreign keys in a sharded database can become a significant development cost to the service.

Rebalancing is an issue that arises when the sharded schema has to be changed. It could happen because the company grew over time and it has different needs. In such situations data has to move according to the new schema over different servers and the system has to go down time.

2.1.5 CAP Theorem

Besides the problems described in Section 2.1.4, the Brewer's conjecture says it is possible to achieve at most two out of three desired properties in a distributed system: consistency, availability and partition tolerance. This conjecture has been formalized and proved by Gilbert and Lynch [22] and it is well known as the *CAP theorem*. In such theorem the properties are reported as follow:

> *"[With consistency] there must exist a total order on all operations such that each operation looks as if it were completed at a single instant."*
>
> *"[With availability] every request received by a non-failing node in the system must result in a response."*
>
> *"[With partition tolerance], the network will be allowed to lose arbitrarily many messages sent from one node to another."*

An example of a network partition is when two nodes cannot talk to each other, but there are clients able to talk to either one or both of those nodes. In a distributed system sharding the database makes the system tolerant to networks partitioning. The choice is then between availability and consistency. In a system which is not available, a request could not receive a response and it can create significant real-world problems. Therefore, NoSQL databases relax consistency in favor of availability (Figure 2.1) and, rather than supporting full ACID, they rely upon different consistency models [19, 34, 20, 24, 16, 17].

2.1.6 Changing paradigm

CAP theorem, denormalization, data replication, increasing complexity in query execution and schema constraints are trade-offs made by DBMS when it comes to deal with a growing database. DMBS were not meant to address these issues at the beginning, but the expertise on these systems and the architecture legacy pushed them to fit under those new constraints.

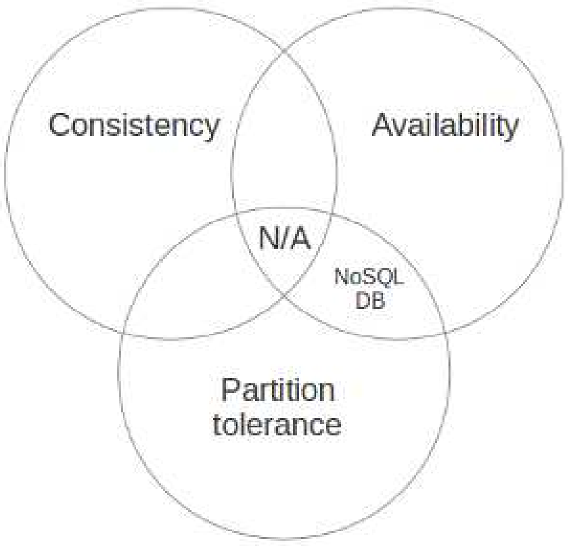


Figure 2.1: CAP Theorem: in a distributed system it is possible to achieve at most two out of three desired properties: consistency, availability and partition tolerance. NoSQL databases relax consistency property.

With the coming of Big Data a paradigm change was required. A new way in storing and querying data emerged to tackle these limits and the applications were grouped under the term NoSQL. Here we will focus on the technology described in two Google papers, regarding the Google file system [21] and MapReduce [14], and their open source implementation Apache Hadoop.

In the following sections we describe into detail Apache Hadoop and Apache Mahout.

2.2 Apache Hadoop

Apache Hadoop is a scalable fault-tolerant distributed system for data storage and processing (open source under the Apache license). It is composed of two main subsystems: *Hadoop Distributed FileSystem* (HDFS) and MapReduce. In this section we describe the whole architecture and the HDFS component while MapReduce is explained in detail in Section 2.3. Figure 2.2 shows how they relate to others tools in the Apache Hadoop ecosystem.

The following documentation about Apache Hadoop components are related to software version 1.1.2.

[2]http://blogs.ejb.cc/archives/4290/hadoop-technical-manuals-a-the-hadoop-ecosystem/tumblr_lbbwggcer71qappj8

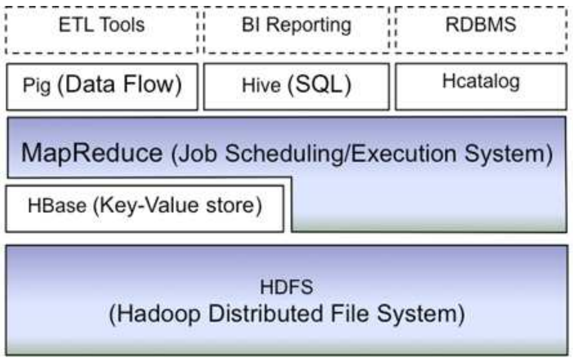


Figure 2.2: Apache Hadoop stack. Apache HBase improves and provides new functionality to HDFS. High-level tools such as Apache Hive, Apache Pig or Hcatalog rely on HDFS as storage system and they eventually take advantage of MapReduce to query data[2].

2.2.1 HDFS Overview

HDFS is a distributed filesystem designed to store a very large amount of data (terabytes or petabytes) and to provide fast, scalable access to it. Computational and storage power should scale according to the number of machines in the cluster, the more the nodes the more clients are served. Reliability is a major concern and if individual machines in the cluster malfunction, data should still be available.

These design choices let HDFS be very scalable, but some trade-off were made with this architecture:

- HDFS is optimized to perform long sequential streaming reads from files, while it is not for random reads.
- Caching is not adeguated since the overhead is great enough that data should be re-read from HDFS source.
- Updating operations to existing files are restricted to appending only.

As HDFS is not a native Unix filesystem, standard Unix file tools do not work. However Apache Hadoop provides a set of command line utilities to interact with the environment, including file and filesystem management and cluster monitoring.

2.2.2 HDFS architecture and Data Replication

Each file is stored in HDFS as a *metadata* file and a collection of *blocks*. While blocks store the content, the metadata file records blocks location and replication. Default configurations set the block size at 64MB and the replication parameter at three, which means that each block is stored three times across the cluster. For performance or reliability reasons these parameters could be tuned by means of xml configuration files (Apache Hadoop website provides a full customizable parameter list[3]).

The HDFS architecture in Figure 2.3 illustrates the master-slave solution to handle nodes availability, workload balancing and replication. The *NameNode* is the master component of the architecture and it stores metadata information, such as where each block is stored and how many times the block is replicated, and tracks *DataNodes*, which actually store the data. If it detects one failure it will restore the lost blocks and create the missing copy. If the NameNode fails, the *BackupNode* (also called *Secondary NameNode*) takes its place.

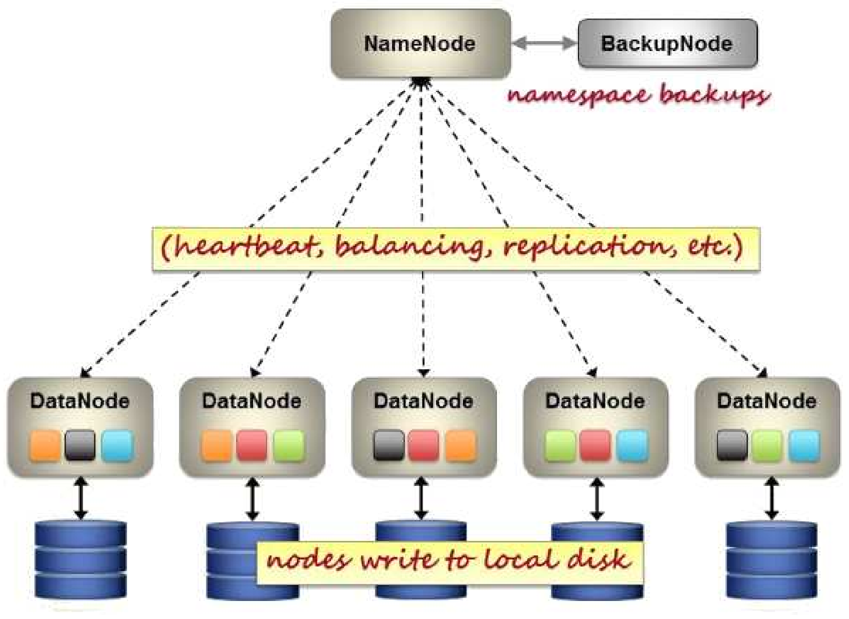


Figure 2.3: HDFS architecture. The NameNode manages DataNodes in storing files. In this picture a file has been partitioned in 5 blocks (not represented) and each block is depicted in a square box with a specific color and stored three times across the cluster[4].

[3]http://hadoop.apache.org/docs/r1.1.2/hdfs-default.html

[4]http://simranjindal.com/2011/10/17/remote-attendees-reflections-sqlpass-2011-day-3-keynote-by-dr-david-dewitt/

2.2.3 Robustness

The three common types of failures are NameNode failures, DataNode failures and network partitions.

The NameNode, which manages the activities in the cluster, faces both DataNode failures and network partitions using *heartbeat* messages. During normal operations DataNodes send lightweight messages (heartbeats) to the NameNode to confirm that it is operating and the blocks it hosts are still available. The default interval is three seconds. If there is not any new incoming heartbeat after a timeout of ten minutes, the NameNode considers the DataNode to be out of service and it starts the recovering procedure of the lost data blocks, replicating those blocks from and to the available DataNodes. Heartbeats also carry information about the general activity and storage status of the DataNode. These statistics are used for the NameNode block allocation and load balancing decisions. Finally as already mentioned, if the NameNode fails the BackupNode takes its place until it recovers.

Because of possible faults in storage devices, network or unreliable software, data fetched from DataNodes could arrive corrupted. A checksum (see Glossary) system is implemented to recognize such problems. For each stored block the DataNode creates an hidden checksum file. When a HDFS client requests a block it receives both the content and the associated checksum file. If there is not match between the local calculated checksum and the one fetched from the DataNode, the client can opt to retrieve that block from another DataNode that has a replica of that block.

2.2.4 Web interface

Apache Hadoop provides a web interface [33] to monitor the health of the cluster and the current status of submitted job through logs.

By default Namenode is reachable using port 50070 and from its interface it is possible to browse the HDFS filesystem or determine the storage available on each individual node.

The JobTracker (Section 2.3.9) is reachable at port 50030. Here, detailed information about the ongoing and completed MapReduce job is available, such as which node performed which tasks and the time or resources required to complete each task.

2.3 MapReduce

MapReduce is a programming paradigm designed to analyze large volumes of data in a parallel fashion. Its goal is to query data in a scalable way, typically stored in HDFS, without any constraints about the cluster size. Architectures such as MPI (Section 2.3.1) where data is shared arbitrarily between nodes, for synchronization for instance, are not reliable because the overhead due to the network traffic could drammatically affect perfomance. Programs written in MapReduce are able to scale with the cluster size, i.e. the same program will properly run with both tens and hundreds of nodes.

In Section 2.3.1 we compare MapReduce with other parallel paradigms. In later sections we describe its most important features related to the development of the master thesis, such as core components, data flow and some advanced components. For a fully detailed description of the paradigm, we suggests books [33, 60] as references.

2.3.1 Parallel paradigms

To understand the differences between MapReduce and Pthreads (POSIX standard for threads [49]) we consider the two systems they rely on: HDFS and POSIX[5] respectively. The latter is a set of standards specified by IEEE for maintaining compatibility between operating systems. In the way MapReduce paradigm works, all the operations on files supported by POSIX are not required. HDFS does not support file modification after they are closed and writes to a single file by multiple clients. In its workflow, MapReduce always creates new files and does not change contents of existing files. Furthermore, Pthreads version lacks fault tolerance and load balancing[6].

MapReduce has been compared to Message Passing Interface (MPI)[7] because they both provide distributed programming environments for cluster level parallelism [37]. MPI is the dominant model used in high-performance computing but its current implementations do not support easy fault tolerance[8], unlike Apache Hadoop and its HDFS which provide data redundancy as core feature. Other differences between the two paradigms are reported in Table 2.1[9].

Finally, OpenMP[10] completely differs from MapReduce since it relies on

[5]http://standards.ieee.org/develop/wg/POSIX.html
[6]http://courses.cs.vt.edu/cs5204/fall10-kafura-BB/Presentations/MapReduce.pdf
[7]http://www.mcs.anl.gov/mpi
[8]http://www.open-mpi.org/faq/?category=ft
[9]http://courses.cs.vt.edu/cs5204/fall10-kafura-BB/Presentations/MapReduce.pdf
[10]http://openmp.org/wp/

	MPI	**MapReduce**
What they are	General parallel programming paradigm	A programming paradigm and its associated execution system
Programming model	Message passing between nodes	Restricted to MapReduce operations
Data organization	No assumptions	Files can be sharded
Execution model	Nodes are indipendent	Map/Shuffle/Reduce
Usability	Difficult to debug	Simple concept, could be hard to optimize
Key Selling Point	Flexible to accomodate various applications	Flow through large amount of data with commodity hardware

Table 2.1: MPI and MapReduce comparison

a shared memory architecture instead of a distributed system.

2.3.2 Data type

MapReduce deals with every kind of data type, such as integer, text, binary (i.e. image and video) or even custom defined ones. In the following examples we consider text only for better explaining concepts.

In Apache Hadoop, each data type is handled through a wrapper class, because wrappers implement functionalities to efficiently manage data throughout the process. Examples are (data type – wrapper) *long* – *LongWritable*, *int* – *IntWritable* and *String* – *Text*.

2.3.3 Core components

MapReduce programs transform lists of input data elements into lists of output data elements. This process happens twice in a program, once for the *map* step and once for the *reduce* step. Those two steps are executed sequentially, the reduce begins when the map is completed.

In the map step the data elements are provided as list of key-value objects. Each element of that list is loaded, one at a time, into a function called *mapper*, which transforms the input and outputs any number of *intermediate* key-value objects. It is worth stressing that the original data is not modified and the mapper output is a list of new objects, because in DBMS database content changes over time, while with MapReduce and according to Section 2.2.1 input does not change.

In the reduce step, intermediate objects that share the same key are grouped together and they are the input of a function called *reducer*. The reduce function is invoked as many times as the number of different keys and its value is an iterator over the related grouped intermediate values.

MapReduce input typically comes from files loaded into HDFS. These files are distributed across all nodes. Mappers and reducers run on many or all of the nodes in the cluster in a isolated environment, i.e. each function is not aware of the other ones and they their task is equivalent in every node. Each mapper loads the set of files local to that machine and processes it. That design choice allows the framework to scale without any constraints about the number of nodes in the cluster.

2.3.4 Word count example

The common example reported in books and online to explain a MapReduce program is the word count example[11]. The purpose of the program is to list unique words in a document corpora and shows how many times each word is present. In Listings 2.1 and 2.2 we provide the pseudo code for the mapper and reducer respectively.

Listing 2.1: Word count mapper

```
mapper(key, value) {
  // key is not important
  List words <- splitAndNormalizeLine(value);
  foreach(word in words) {
    output(word, 1);
  }
}
```

Listing 2.2: Word count reducer

```
reducer(key, iterator<values>) {
  counter <- 0;
  foreach(value in values) {
    counter <- counter + 1;
  }
  output(key, counter);
}
```

[11]http://wiki.apache.org/hadoop/WordCount

In this example the mapper input is a line in the text document. In key-value terms the key is the line offset in the document and the value is the line itself. The mapper outputs an intermediate key-value object each time a word is read, where the key is the word itself and the value is the constant integer *1*. The reducer groups intermediate objects by word, it sums up how many times the word has been read by mappers and outputs the counting. Function *splitAndNormalizeLine* in Listing 2.1 line 3 splits the line according to an empty space delimiter and normalizes each words, i.e. punctuation is removed and every word becomes lower case.

Other simple and complex examples are described into details in books [33, 60], such as the patent citations, average, reduce-side join.

2.3.5 MapReduce data flow

This section introduces the MapReduce data flow from a high level view. In Figure 2.4 data in the storage system is split in small chuncks, depicted as grey rectangular boxes, each one is the mapper input in key-value format. In this example three mappers are shown that transform the input in intermediate output objects, yellow boxes.

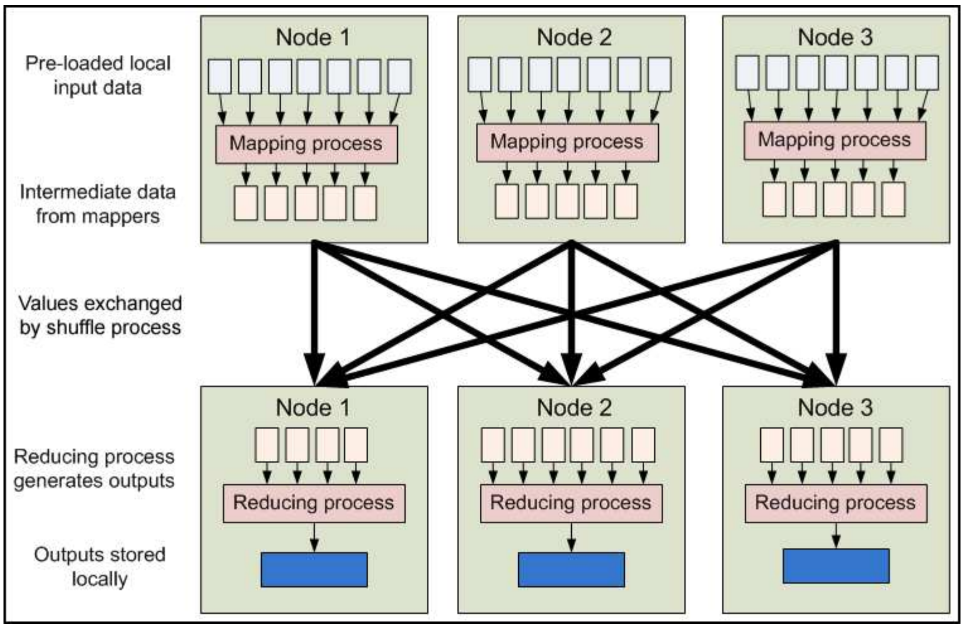

Pre-loaded local input data
Intermediate data from mappers
Values exchanged by shuffle process
Reducing process generates outputs
Outputs stored locally
Node 1
Node 2
Node 3
Mapping process
Reducing process

Figure 2.4: High level view of MapReduce data flow[12].

When the mapping phase is completed, the intermediate objects must

[12]http://developer.yahoo.com/hadoop/tutorial/module4.html

be exchanged between machines to send all values with the same key to a single reducer. The reduce tasks are spread across the same nodes in the cluster as the mappers. This is the only communication step in MapReduce, completely handled by the Apache Hadoop platform and guided by the different keys associated with values.

Though the user does not manage the network traffic, he can customize its behavior. Between map and reduce steps, Apache Hadoop lets the user customize how data is partitioned across nodes and how it is sorted before being read from reducers. These steps are called *partition* and *sort*, respectively.

The partition is managed by the *Partitioner* class, which pseudo code interface is reported in Listing 2.3. The *getPartition* function receives the intermediate key-value objects, one at a time, and the number of partitions to split the data across. It has to return an integer value in the range between 0 (included) and *numPartitions* (excluded). There are usually as many partitions as nodes. The default partitioner computes a hash value for the key and assigns the partition based on this result.

Listing 2.3: Partitioner interface

```
interface Partitioner {
  getPartition(key, value, numPartitions);
}
```

Custom partitiong is useful to address *skewness* of data. An unbalanced distribution of intermediate objects (based on keys) will affect MapReduce performance, because the reducer with most workload will slow down the entire job while the already completed reducers will stay idle. If the unbalanced distribution of keys is known a priori, the custom partition could split data according to that knowledge.

For example, suppose there are five partitions and keys are in the range $[0, 9]$. With balanced distribution each partition takes data related to two keys and the workload is equally distributed. An example of data skew is when intermediate objects related to key 0 is one fifth of the whole data (instead of one tenth). To better distribute the workload across reducers, one partition is in charge to load objects with key 0 only, while the others do the rest.

After partitioning, the next step is sorting. The set of intermediate keys on a single node is automatically sorted by Apache Hadoop before they are presented to the reducer. Several MapReduce jobs do not require an ordered input to reducers and it will be more efficient to remove that step to increase performance. The sorting operation is not about enforcing the sorted order of the reducer input, but sorting is an efficient way to group all records of the same key together [33].

A custom sorting is injected by extending the class of the key to *Writable-Comparator* class and implement *compare* function. In Listing 2.4 we report the pseudo class implementation.

Listing 2.4: Extending WritableComparator class

```
class MyKey extends WritableComparator {
  compare(obj1, obj2) {
    cmp = 0;

    // compare obj1 and obj2 according to their
    // definition and set cmp variable

    return cmp;
  }
}
```

2.3.6 Combiner

The *combiner* is an optional functionality in MapReduce. It has been introduced to process and aggregate intermediate objects before the partition step begins to reduce the network traffic across nodes. In Figure 2.5 is shown the combiner in the MapReduce data flow.

The combiner interface is the same of the reducer one, but it operates only on data generated by one machine. In the Word Count example the combiner could be applied to decrease the amount of intermediate objects sent across the network. In one partition, suppose the mapper outputs 10 times the object ("house", 1), meaning the word "house" has been read ten times in the locally stored data. Without the combiner all those objects are sent to another node, while with the combiner the objects are first aggregated in ("house", 10) and then only a single object is sent.

2.3.7 Distributed cache

DistributedCache [33] is a facility provided by the MapReduce framework to share read only files globally with all nodes on the cluster. Files are copied to all local machines before the job runs and they can be texts, libraries or executable code in general.

[13]http://developer.yahoo.com/hadoop/tutorial/module4.html

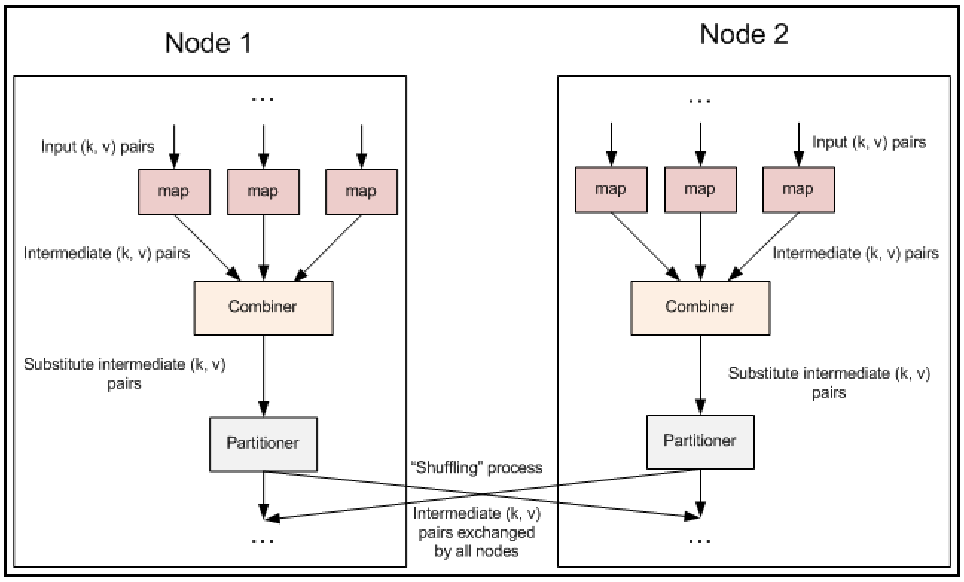

Node 1
Node 2
Input (k, v) pairs
map
Intermediate (k, v) pairs
Combiner
Substitute intermediate (k, v) pairs
Partitioner
"Shuffling" process
Intermediate (k, v) pairs exchanged by all nodes

Figure 2.5: Combiner step inserted into the MapReduce data flow[13].

2.3.8 Reading and writing

In Figure 2.6 a more detailed data flow is shown, introducing the interaction between MapReduce and HDFS related to inputs and outputs.

Files in the HDFS have typically size of gigabytes of terabytes. They are split in smaller *chunks*, called in Apache Hadoop terminology *input splits*, which are processed in parallel. Each chunck is the input of a mapper function, so it should not be both too large to outfit the main memory and too small that the overhead of starting and stopping the processing of a split becomes a large fraction of the execution time. The default chunk size is 64MB.

Input splits are a logical division of the file in HDFS, in relatively rare situations it could happen a mapper input is physically partitioned into two or more nodes. The process of fetching the data is managed by Apache Hadoop and it is transparent to the user, by the way some overhead occurs.

InputFormat provides a way to read the file (i.e. there is no constraints about the file format, so a custom reader has to be defined if necessary), defines how files are broken into chuncks and how to load the data from its source and convert it into (key, value) pairs suitable for reading by the mapper, by means of *RecordReader* (RR). Popular InputFormat classes are reported in the following list (taken from [33]):

[14]http://developer.yahoo.com/hadoop/tutorial/module4.html

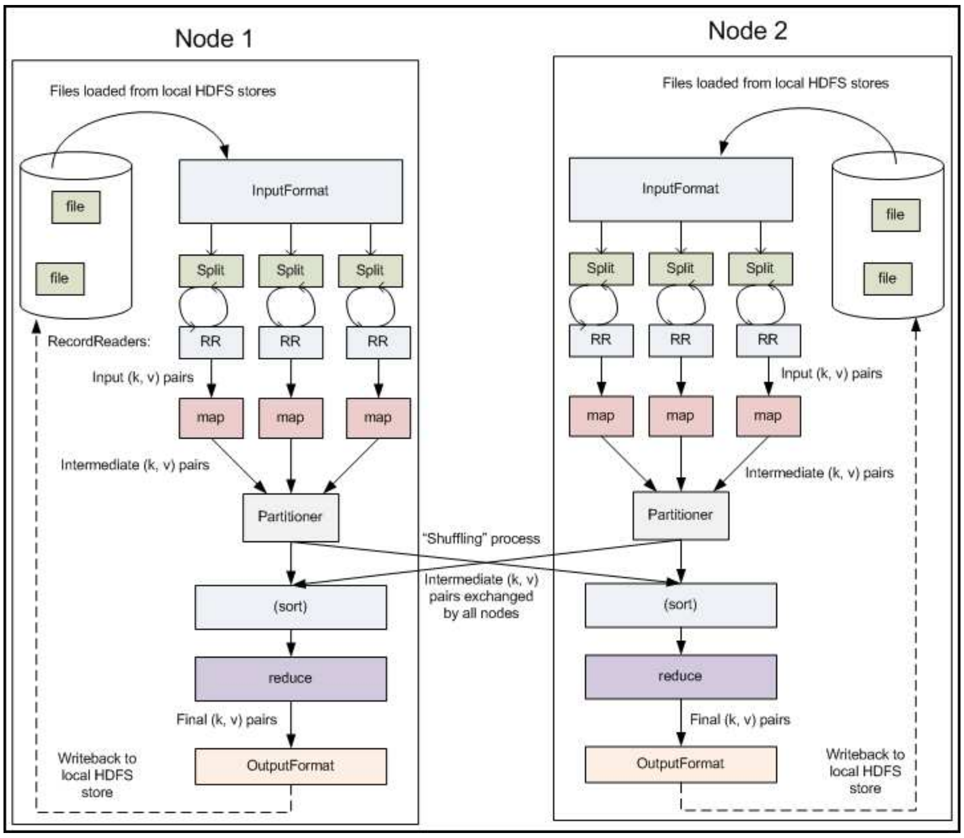


Figure 2.6: Detailed Hadoop MapReduce data flow[14].

- *TextInputFormat*: each line in text files is a record read. Key is the byte offset of the line, and value is the content of the line.

- *KeyValueTextInputFormat*: each line in text files is a record read. The first separator character divides each line. Everything before the separator is the key, everything after is the value. The separator is set customizable and the default is the tab character.

- *SequenceFileInputFormat<K, V>*: an InputFormat for reading SequenceFiles, which is an Apache Hadoop specific compressed binary file format. It is optimized for passing data between the output of one MapReduce job to the input of some other MapReduce job. Key and value are user defined.

- *NLineInputFormat*: same as TextInputFormat, but each split is guaranteed to have exactly N lines. The number of lines is customizable and its default value is one.

Apache Hadoop defines outputs of MapReduce jobs by *OutputFormat* and it works likewise the InputFormat. OutputFormat validates the output-specification of the job, i.e. it checks that the output directory does not already exist and provides a way to write out job output files, which are typically stored in the HDFS. Popular OutputFormat classes are reported in the following list ([33]):

- *TextOutputFormat*: writes each record as a line of text. Keys and values are written as strings and separated by a custom character (tab character is the default one).
- *SequenceFileOutputFormat*: writes the key value pairs in Apache Hadoop SequenceFile format. It works in conjunction with SequenceFileInputFormat.
- *NullOutputFormat*: outputs nothing.

2.3.9 MapReduce architecture in Apache Hadoop

In Section 2.2.2 we described how data is managed by a master-slave architecture by means of a NameNode and several DataNodes. The same approach is applied to MapReduce jobs, Figure 2.7. Each MapReduce job submitted by a client is managed by the *JobTracker*, which determines the execution plan, assigns nodes to different tasks and monitors the execution.

The *TaskTracker* is in charge of executing the tasks the JobTracker assigns. In the cluster there is one master JobTracker only and as many TaskTracker as the number of slaves. Since multiple MapReduce jobs could run simultaneously on the cluster, the JobTracker assigns tasks from different jobs to the same TaskTracker.

The communication protocol design between JobTracker and TaskTracker aims both to monitor the jobs status and to recover after a node failure. The principle is the same described in Section 2.2.3, related to the robustness of HDFS architecture. The JobTracker receives periodic heartbeats from each TaskTracker. If JobTracker does not receive any heartbeat from one TaskTracker after a period of time (1 minute by default) it will assume the TaskTracker in question has failed and it will resubmit the corresponding tasks to other nodes in the cluster.

Having described how Apache Hadoop handles data and jobs, the overview of the entire architecture is in Figure 2.8.

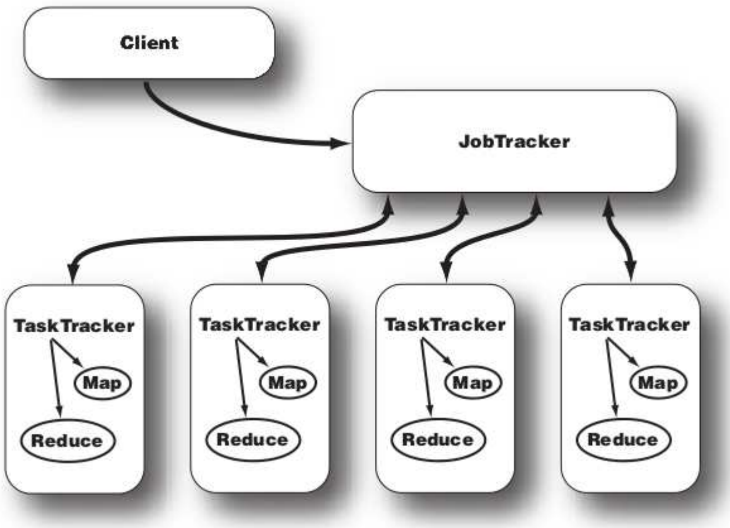


Figure 2.7: JobTracker and TaskTracker interaction. The JobTracker partitions the workload of the client submitted job to different TaskTrakers [33].

2.3.10 Language support

Java is the native language in Apache Hadoop to program in MapReduce, however *Apache Hadoop Streaming*[15] is a versatile tool that interacts with Unix environment and allows the developer to use arbitrary programs as mapper and reducer, receiving and sending data through stdin and stdout (see Glossary) in text format only. In a single machine Unix environment that process is like executing Unix commands in pipeline. Listing 2.5 shows the pseudo code.

Listing 2.5: Pseudo code using Unix command line notation

```
cat [input_file] | [mapper] | sort | [reducer] >
    [output_file]
```

With Apache Hadoop Streaming the job is distributed across the cluster and programs can be written in any script language such as bash, python, perl, or another language of choice, provided that the necessary interpreter is present on all nodes in the cluster.

[33, 60] contain examples of Apache Hadoop Streaming usage from command line, mappers and reducers written in script languages.

[15]http://hadoop.apache.org/docs/stable/streaming.html

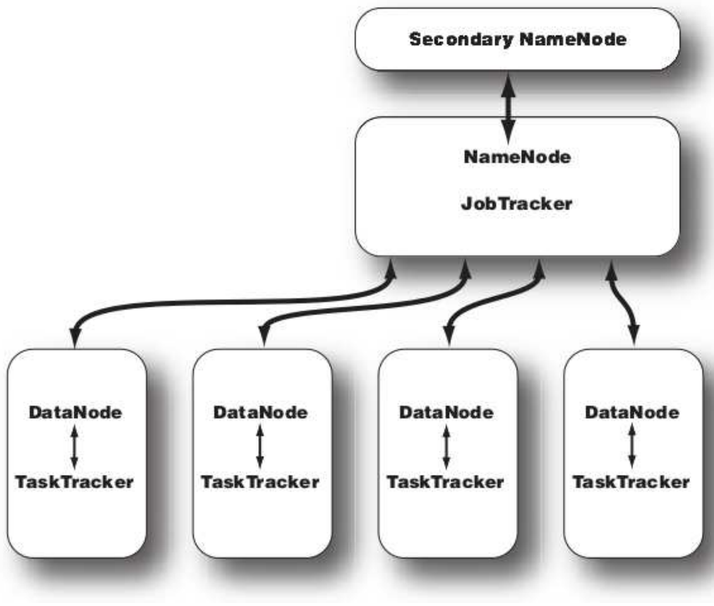


Figure 2.8: Apache Hadoop master-slave architectures for handling data and MapReduce jobs [33].

2.4 Analytics with Big Data

In Big Data scenario, data analysts used to work with DBMS should rewrite the algorithm implementations in MapReduce fashion. From a business perspective, besides the high cost required both in employee training and in converting data analytics tool in MapReduce, all the technical expertise (especially the SQL-like) would be lost.

High-level tools purposefully emerged to overcome some Apache Hadoop shortcomings, allowing complex business tasks and increasing productivity in MapReduce. In the following sections four products are described, each one provides analytics in different ways. Apache Mahout provides machine learning tools for Big Data and it is described in detail in Section 2.5 because it is a core component of the master thesis.

2.4.1 Apache HBase

Apache HBase is a type of NoSQL database (in Section 1.1 it has been listed under Column Family Stores group). It provides a way to store data, rather than to create a database, because it lacks many of the features available in DBMS, such as typed columns, secondary indexes, triggers, and advanced query languages. Considering the trade-off made with HDFS in Section

2.2.1, some feature of note are:

- It is built on top of HDFS and provides fast record lookups (and updates) for large tables.
- It supports caching and bloom filters [4] for high volume query optimization.
- Tables are distributed on the cluster via regions, and regions are automatically split and re-distributed as data grows.
- MapReduce could use Apache HBase as both source and sink.
- It provides strongly consistent reads/writes and random, realtime read/ write access to data.

Apache HBase is a middle layer between Apache Hadoop and other high-level softwares, which actually deliver data analytics tools. It could be accessed by means of Java API, REST/HTTP, Apache Thrift[16] or Apache Hive (Section 2.4.2).

2.4.2 Apache Hive

Apache Hive[17] is a data warehouse system for the analysis of large datasets stored in Apache Hadoop compatible filesystems. The core component is the SQL-interface called *HiveQL*, which allows the developer to abstract relational-like structure on top of non-relational or structured data. The engine could be improved by plugging in custom Mappers and Reducers, when it is inconvenient or inefficient to express this logic in HiveQL.

Figure 2.9 introduces Apache Hive architecture (green blocks) and its relation with Apache HBase (golden blocks). The software can be accessed through command line, a web interface or by means of an Apache Hive Client, linked to Thrift server:

- The command line is a shell utility which can be used to run Apache Hive queries in either interactive or batch mode.
- The web interface is an alternative to using the Apache Hive command line. It visualizes metadata information such as database and tables.

[16]http://thrift.apache.org/

[17]http://hive.apache.org/

[18]http://www.slideshare.net/hortonworks/integration-of-hive-and-hbase-12805463?from_search=12

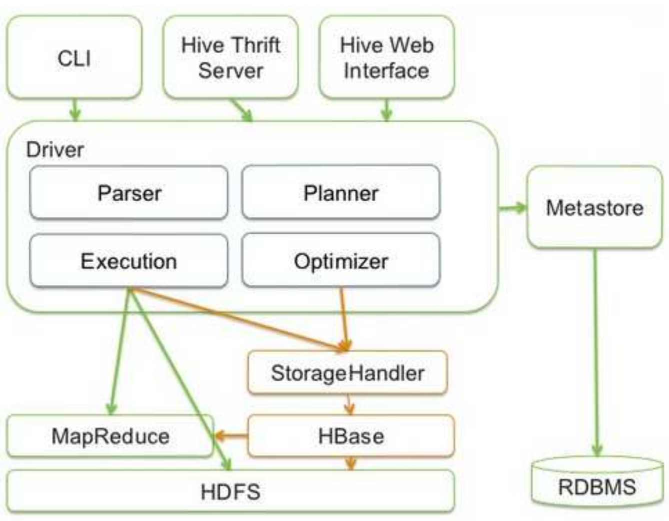


Figure 2.9: Apache Hive architecture and its integration with Apache HBase[18].

- The Apache Hive Client supports different kinds of clients, such as JDBC, Python, PHP, etc...

The Driver is the component which receives the queries. The subcomponents Parser, Planner, Execution and Optimizer work together to parse the query, do semantic analysis on the different query expressions and generates an execution plan, using metadata information retrieved from the metastore. The plan is a directed acyclic graph (DAG) of stages, composed by MapReduce jobs and HDFS commands, which dependencies are managed by the execution engine.

Apache Hive is introducing Apache Hadoop technology to a wider audience of analysts and other nonprogrammers. As of August 2009, Facebook counts 29 percent of its employees as Apache Hive users, more than half of whom are outside of engineering, i.e. they come from distinct groups like sales, human resources, and finance[19].

2.4.3 MLbase

MLbase [32] aims to simplify accessibility to machine learning algorithms in a distributed environment. The system itself manages load balances, data partitioning among cluster nodes and provides built-in common algorithms such as SVM [13]. It is possible to extends the algorithm set through a custom high level language.

[19]https://www.facebook.com/notes/facebook-data-science/distributed-data-analysis-at-facebook/114588058858

There are several systems that already provide the functionalities described above [32], such as Apache Hadoop combined with Apache Mahout as detailed in Section 2.5, but major MLbase contribution is the *optimizer* component which selects automatically the best model for the committed task with related parameters. MLbase uses down-sampled data to speedup the evaluation of different learning algorithms applicable to the specific task. After exploration, the best model is trained with the larger dataset.

The system is still an early prototype, but the first public version will have tools for common machine learning tasks such as collaborative filtering, dimensionality reduction and data visualization.

2.4.4 R programming language

R[20] is a free software programming language and a software environment for statistical computing and graphics. The R language was built by statisticians for developing statistical software and data analysis. According to Bo Cowgill (ex Google employee), R is the most popular statistical package at Google[21].

R environment was not built for Big Data analytics because it is memory-bound, but the combination of R and Apache Hadoop could be a solution to bring analytics in Apache Hadoop. There are some challenges that need to be addressed. First, analytics is often an iterative process. This is what makes R such a powerful tool, a good environment for performing such analysis. Apache Hadoop on the other hand is batch oriented where jobs are queued and then executed, it may take minutes or hours to run these jobs. Second, R is designed to have all of its data in memory, while programs in Apache Hadoop (MapReduce jobs) work independently and in parallel on individual data slices.

The most common way to link R and Apache Hadoop is to use HDFS (potentially managed by Apache Hive or Apache HBase) as the long-term store for all data, and use MapReduce jobs to encode, enrich, and sample datasets from HDFS into R. Data analysts can then perform complex modeling tasks on a subset of prepared data in R.

For a full integration between R and MapReduce, the R landscape provides some useful libraries and it is possible to group them in two categories.

In the first group the goal is providing a simple and usable interface that allows specification of both map and reduce functions in R. R programmers might have to rethink the approach to how algorithms can realized

[20]http://www.r-project.org/

[21]http://dataspora.com/blog/predictive-analytics-using-r/

and implemented, but the underlying optimization should justify the additional effort. In R, functions or calculations that fit nicely into MapReduce model are the *apply* operation family, quantiles, crosstabs and stochastic calculations (like Monte Carlo simulations). R libraries such as RHadoop[22], RHIPE[23] and ORCH[24] belong to this category.

In the second group, the MapReduce execution is fully transparent to the developer and RHive[25] belongs to this categoty. RHive is an R extension facilitating distributed computing via Apache Hive query and provides an easy to use execution of R functions and objects through Hive Query Language.

2.5 Apache Mahout

Apache Mahout is a Java library which goal is to provide scalable machine learning algorithms. In this section we explore high level functionalities available for end-users and reusable code to create custom tools. We considered Apache Mahout stable version 0.7.

2.5.1 Apache Mahout and Apache Hadoop

Apache Mahout primary goal is to provide scalable machine learning libraries. It can run in standalone mode, but it can take advantage of Apache Hadoop if both software are running in the same machine or cluster.

All implemented algorithms run in a single machine, and *some* of them are implemented in distributed mode using MapReduce paradigm. In the following sections we introduce a detailed description of which of them are in both fashions or just the sequential one.

2.5.2 Tools and algorithms

The documentation related to the functionalities in Apache Mahout is included in the book "Mahout in Action" [50], in the online documentation[26] and the source code, downloadable from the website[27]. The functionalities available could be grouped in four categories:

[22]https://github.com/RevolutionAnalytics/RHadoop
[23]http://www.datadr.org/
[24]https://blogs.oracle.com/R/entry/introduction_to_oracle_r_connector
[25]http://cran.r-project.org/web/packages/RHive/
[26]https://cwiki.apache.org/confluence/display/MAHOUT/Mahout+Wiki
[27]http://mahout.apache.org/

Classification	
Algorithm	**MapReduce**
Logistic regression	no
Adaptive logistic regression	no
Naive bayes	yes
Random forest	yes

Table 2.2: Classification algorithms in MapReduce on Apache Mahout.

- *preprocessing*: the majority of the algorithms require the dataset in an intermediate format to run properly. A raw dataset is converted in a *vectorized* format, where data is organized in a vector format for better performance.

- *algorithms*: algorithms are described in Appendix A and they are grouped in classification, clustering and recommendation.

- *postprocessing*: these tools convert the vectorized data or the output of Apache Mahout jobs in a human-readable format.

- *utilities*: a pool of different tools that helps the development and management of custom tools.

The end user can exploit all functionalities as a black box, because each tool can be invoked by a command line interface. Executing an algorithm from the command line with the related parameters properly set is the easiest way to use Apache Mahout. The online documentation and the software itself describe a detailed list of parameters for each functionality.

More complex and custom functionalities could run on top of Apache Mahout. In such situations, the user has to develop Java code and leverage the library interface. Every functionality has an interface, so a set of algorithms (even custom ones) could be chained sequentially to run a complex job.

A detailed description of Apache Mahout tools is reported in Appendix A, while in Tables 2.2, 2.3, 2.4 and 2.5 we summurize which functionalities are implemented in MapReduce fashion, besides the sequential one. It is worth noticing that there are few classification algorithms in MapReduce fashion.

Clustering	
Algorithm	**MapReduce**
Canopy	yes
Top down	yes
Dirichlet	yes
Eigencuts	yes
Fuzzy k-means	yes
K-means	yes
MinHash	yes
Mean Shift	yes
Spectral k-means	yes

Table 2.3: Clustering algorithms in MapReduce on Apache Mahout.

Recommendation	
Algorithm	**MapReduce**
K-nearest neighbors	no
Threshold-based neighborhood	no
Item Based Recommendation	yes

Table 2.4: Recommendation algorithms in MapReduce on Apache Mahout.

Other	
Tools	**MapReduce**
Frequent Pattern Mining	yes
ALS-WR Factorization Matrix	yes
Row Similarity	yes
SVD	yes
SSVD	yes
Viterbi	yes
CVB	yes
Baum Welch	yes

Table 2.5: Other tools in MapReduce on Apache Mahout.

2.5.3 Alternatives to Apache Mahout

There are three other frameworks worth mentioning, as Apache Hadoop and Apache Mahout alternatives:

- AMPLab researchers, at UC Berkeley, released *Spark* [62] which was developed to reduce latency data sharing in iterative algorithms, common in machine learning and data mining fields. The research group achieved the goal introducing a memory abstraction called Resilient Distributed Datasets, which is extensively described in [62].

- *Graphlab* [39], developed at UC Berkeley, is a graph-based, high performance, distributed computation framework written in C++. The goal was to achieve excellent parallel performance on large scale problems.

- *Vowpal Wabbit* [1]: it is a project started at Yahoo! Research and continuing at Microsoft Research. The general goal is to create a very fast, efficient, and capable learning algorithm.

Even though they overcome in different ways some issues related to Apache Hadoop, they are not fully compliant with the research purposes: Spark is a completely separate codebase from Apache Hadoop, Graphlab implements a modified version of MapReduce while Vowpal Wabbit is focused on online learning. We chose Apache Mahout as a machine learning library because it is a general framework for MapReduce machine learning algorithms and it can be deployed on top of Apache Hadoop, leveraging the full scalability it provides.

Chapter 3

Feature Selection algorithms in MapReduce

"[Feature selection] objective is to select the minimal subset of features according to some reasonable criteria so that the original task can be achieved equally well, if not better"

Huan Liu, author of *Feature Selection for Knowledge Discovery and Data Mining*

The prediction task is heavily affected by the quality and quantity of features it relies on. Considering irrelevant features will introduce noise, possibly leading to *overfitting* and increased computational complexity. Conversely, excluding important features can deprive the algorithm of important information.

The goal of feature selection algorithms is to select the best subset of features in terms of accuracy or other parameters. This task is more difficult and complex as datasets have high dimensions, which is the standard nowadays in real-world applications. Scaling algorithms in a distributed system is one more concern in the Big Data scenario.

Large scale and high dimension datasets are used in applications such as internet algorithms, computational biology or social link analysis [54]. Traditional single machine algorithms may no longer be feasible to produce models in reasonable time. For these reasons we considered feature selection algorithms in MapReduce for distributed systems, which are not available in popular open source tools such as Apache Mahout (Section 2.5).

In this chapter we introduce feature selection algorithms (Section 3.1), the challenges we faced to reshape and redesign them in MapReduce (Section 3.2) and finally we present our contributions (Section 3.3).

Throughout the document we are coherent regarding the annotation used, which is reported in the Annotation chapter.

3.1 Introduction to feature selection

Before describing feature selection algorithms it is important to introduce some terminology and convention. We consider a dataset as a collection of data organized in a table fashion. An *instance* is the single result of an event we are interested in and it is organized as a table row, while each column defines a measured property of that event and we call it *feature*. In supervised learning there is an additional column that classifies the instance, the *target feature*, where its possible values are called *target classes* (or *classes* only). For simplicity, we consider this situation only from now on.

For example, in the healthcare domain, the physician gathers data from its patients about a disease and related symptoms, where each instance has data related to a single patient. Each feature is linked to a single symptom and the target feature collects whether the patient has (or not) the tracked disease. The target classes in this example are two: yes, the patient has the disesase, no, he has not. Based on the collected data, the goal of the physician is to predicted whether a new patient has (or not) the disease, given a set of symptoms. The algorithm that helps in the classification process is called *induction algorithm*.

There are some domains of interests with hundreds to tens of thousands of features. All those features were tracked to better understand the behavior of the target class, but some of them could provide no useful information and actually harm during the prediction analysis. Applying feature selection techniques to datasets with large number of features has some benefits: facilitate data visualization and data understanding, reducing storage requirements, reducing training and utilization times, defying the curse of dimensionality and overfitting to improve prediction performance [25]. Domains which exploit feature selection techniques include text processing of internet documents, gene expression array analysis and combinatorial chemistry.

Feature selection algorithms could be applied both in supervised and unsupervised learning. In the first case, we want to determine feature relevance according to the target class, while in the second case we do not have classes and the target concept is usually related to the structures of the data. In reinforcement learning problems, feature selection is used for instance to determine which subset of all inputs fed to the agent should be included to generate the best performance.

3.1.1 Dependence and correlation

In statistics, *dependence* refers to any relationship between two random variables or two sets of data. In probabilistic terms, those variables are indipendent. Measuring the relationship between features and the target objective is a way to discover the underlying structure of the data and build models for prediction. A frequently used quantity to measure dependence is *correlation* [36].

The *Pearson correlation* is the most common among all the other types of correlation [5]. It is a number between -1 and 1 that meausures the tendency of two series of numbers, paired up one-to-one, to move together. For every pair, if both numbers are either very high or very low at the same time, then the correlation is close to 1; if they are high in module but differ in sign, then they move in opposite direction and their correlation is close to -1. When there appears to be little relationship at all, the value is near to zero.

Formally, Pearson correlation coefficient, ρ, between two random variables is defined as the covariance of the two variables divided by the product of their standard deviations (Formula 3.1).

$$\rho_{X,Y} = \operatorname{corr}(X,Y) = \frac{\operatorname{cov}(X,Y)}{\sigma_X \sigma_Y} = \frac{E\left[(X-\mu_X)(Y-\mu_Y)\right]}{\sigma_X \sigma_Y} \tag{3.1}$$

If we have a series of n measurements of X and Y written as x_i and y_i where i = 1,2,...,n, then Formula 3.2 can be used to estimate the Pearson correlation between X and Y.

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})}{(n-1)\,s_x s_y} = \frac{\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i-\bar{x})^2 \sum_{i=1}^{n}(y_i-\bar{y})^2}} \tag{3.2}$$

Correlation coefficients detect linear dependence only [36]. Other measures were developed to be more sensitive to nonlinear relationships, such as the mutual information.

3.1.2 Mutual Information

Mutual information is a measure of the dependence between two random variables [36]. It is zero if the variables are indipendent, it is large if one is a function of the other. The mutual information is non-negative, i.e. $I(X;Y) \geq 0$, and symmetric, i.e. $I(X;Y) = I(Y;X)$.

Formally, the mutual information of two continuous random variables X and Y can be defined as

$$I(X;Y) = \int_Y \int_X p(x,y) \log\left(\frac{p(x,y)}{p(x)\,p(y)}\right) \tag{3.3}$$

where

- $p(x,y)$ is the joint probability density function of X and Y.
- $p(x)$ is the marginal probability density function of X.
- $p(y)$ is the marginal probability density function of Y.

In the discrete case, the definition is the following:

$$I(X;Y) = \sum_{y\in Y} \sum_{x\in X} p(x,y) \log\left(\frac{p(x,y)}{p(x)\,p(y)}\right) \tag{3.4}$$

where

- $p(x,y)$ is the joint probability distribution function of X and Y.
- $p(x)$ is the marginal probability distribution function of X.
- $p(y)$ is the marginal probability distribution function of Y.

3.1.3 Algorithm categories

Feature selection algorithms navigate the features space looking for subsets of features that best support the given goal. A score function which measures each subset of features is used to compare different possibile solutions. The simplest algorithm is to test each possible subset finding the one which minimises the error rate. This is an exhaustive search of the space, but the total number of possibile subsets is 2^m where m is the number of features. This is computationally intractable for all but the smallest of feature sets. Besides that approach, feature selection algorithm are grouped in three categories [25].

Filter methods for subset selection rank each feature according to some metric and select the highest ranking features. The scoring should reflect the discriminative power of each feature. These methods easily scale to high-dimensional datasets and are fast to compute but they consider features individually only, indeed they cannot detect inter-feature-depencencies. In the XOR example, reported in Figure 3.1, neither the first nor the second feature alone helps to determine the class of the example, only both features together contain enough information about that. Another drawback of filter methods is their total ignorance about the effects of the selected feature subset on the performance of the induction algorithm [31]. Because of these disadvantages, they are often used in the preprocessing step to reduce space dimensionality and then apply more complex algorithms afterwards.

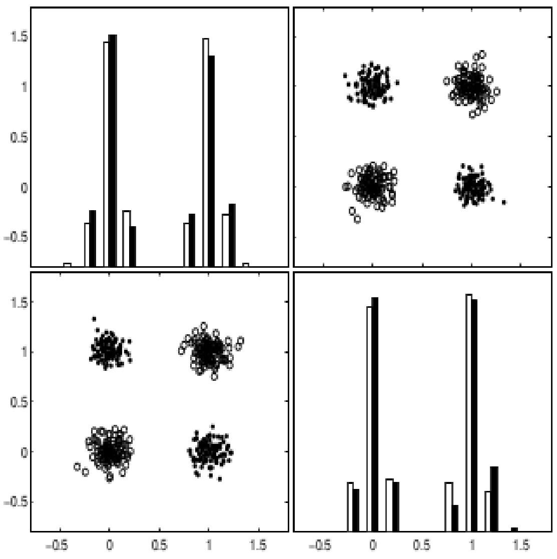


Figure 3.1: XOR example. The conditional density classes overlap when projected to the axes. Therefore, individual variables have no separation power, but variable combination provides good class separability [25].

Wrapper methods take into account the induction algorithm perfomance as a black box, so only the interface is required and no knowledge about the implementation is needed. The role of the induction algorithm is to lead the wrapper in searching the feature space by means of an accuracy measure. In such situations the wrapper could perform the search as a brute force algorithm, exploring all the possible subsets but, as already explained, it is not feasible and some search strategies and heuristics are considered, such as hill-climbing and best-first search [31]. The goal of the search is to find the subset with the highest evaluation, using an heuristic function to guide it, at cost of increasing the complexity of the algorithm and its performance time.

There are three approaches in order to build the feature subset in wrapper methods. *Forward selection* starts with zero (or few) feature selected and at each iteration a new one is added according to an evaluation measure; on the other hand, *backward elimination* begins at the full set of features and deleting those features that helps improving the performance [15, 45].

Combining these previous two approaches yields to *nested subset*, where at each iteration a feature can be added as well as removed from the selection. Stepwise regression [29] is an example of wrapper method.

Embedded methods are less computationally intensive than wrapper methods because feature selection and the induction algorithm cannot be separated. The complexity of the method decreases, but on the other hand the embedded method is specific to the given learning machine. This is a different approach with respect to the previous methods described. Filters do not incorporate any induction algorithm, while in wrappers the feature selection is not integrated into the learning machine implementation.

3.1.4 Minimal Redundancy Maximal Relevance

Minimal Redundancy Maximal Relevance (*mRMR*) [51] is a feature selection algorithm based on mutual information belonging to filter category. As the name suggests, it relies on two metrics: the maximal relevance between candidate features and target class, the minimal redundancy between the selected features.

In maximal relevance, features are ranked in descent order according to their mutual information with the target feature. Given the target class c, for each feature x_i the value $I(x_i; c)$ is calculated and those values are ordered from the largest to the smallest value.

Building a predicting model based on maximal relevance criteria only does not guarantee good performance. As explained in Section 1.3, redundant selected features could affect model performance. mRMR wants to combine both maximal relevance and minimal redundancy criteria.

Definition

The maximal relevance criterion is described as maximizing the mean value of all mutual information values between individual feature x_i, in the feature selected set S, and class c:

$$\max D(S, c), \quad D = \frac{1}{|S|} \sum_{x_i \in S} I(x_i; c) \tag{3.5}$$

The minimal redundancy criterion is described as minimizing the mutual information between the selected features:

$$\min R(S), \quad R = \frac{1}{|S|^2} \sum_{x_i, x_j \in S} I(x_i; x_j) \tag{3.6}$$

Minimal redundancy maximal relevance criterion combines these two contraints with the operator $\Phi(D,R)$ and optimizing D and R simultaneously:

$$\max \Phi(D,R), \quad \Phi = D - R \tag{3.7}$$

Incremental search method can be used to find features defined by $\Phi(.)$. Suppose we already have S_{m-1}, the feature set with $m-1$ features. The task is to select the mth feature from the set $\{X - S_{m-1}\}$. This is done by selecting the feature that maximizes $\Phi(.)$. The respective incremental algorithm optimizes the following condition [51]:

$$\max_{x_j \in X - S_{m-1}} \left[I(x_j; c) - \frac{1}{m-1} \sum_{x_i \in S_{m-1}} I(x_j; x_i) \right] \tag{3.8}$$

3.2 Feature selection in MapReduce

In this section we describe the challenges faced in developing algorithms in MapReduce. In the Big Data scenario, it is important to remark that it is not possible to reuse Java libraries available for computing the algorithm since they require as input the entire vector of feature data, the main memory is usually not big enough to store all the data. Algorithms have to be reshaped using mapper and reducer functions.

Related works focused on developing a parallel programming method that can be applied to many different learning algorithms in a multicore system [10], analyzing the design and the performance of machine learning algorithms [23], and describing a framework in distributed environment with MapReduce for matrix computation [53], tasks commonly used for statistical calculations. Another related work [54] developed a set of feature selection algorithms for MapReduce framework, in which the infrastructure used is based on Google system. Unfortunately, we were not able to implement algorithms explained in [54] because their reduce step read values more than once, which is not possible in Apache Hadoop. To our knowledge, we are the first one implementing feature selection algorithms for Apache Hadoop.

During the implementation step our major challenge faced was properly designing the algorithm. There were several possibile solutions, briefly described as follows:

- It is possible to design the algorithm as a single MapReduce job, where all the information required is outputted and properly tagged from the map step and analyzed in the reduce step. It is not always possibile to follow this approach because the set of all possibile feature subset is exponential to the number of features, i.e. it is not feasible to manage all intermediate objects through the network. The alternative is to use a sequence (or an iteration) of MapReduce jobs.

- The algorithm can be fed with a dataset by rows or by columns. In the latter there is eventually a preprocessing step in which the dataset is trasposed.
- Through Apache Hadoop it is possible to leverage the Distributed-Cache (Section 2.3.7) as a little shared memory across nodes. Here the design choice is to select what information should be shared and in which format.
- Choosing between feeding the algorithm with a dataset either in the original format or optimized for Apache Hadoop (SequenceFile, Section 2.3.8). With the latter there is a preprocessing step in which the data is copied in the SequenceFile format in the distributed storage. The trade-off is between that step and better performance.
- Take advantage of custom data types in handling data.

3.3 Contributions

In this section we describe in detail what are our contributions in feature selection algorithms. We developed a mutual information calculation for MapReduce and two filter feature selection algorithms in MapReduce (Ranking based on Pearson correlation coefficient and mRMR). The Java code is an extension of Apache Mahout and it is publicly available on GitHub[1].

For both algorithm designs, we chose to read the original dataset by rows, so no preprocessing step was required, scalability over rows is achieved in the map step and scalability over columns in the reduce step (this concept is explained in details in the following sections). Related to data types, we opted to use the ones already provided by Apache Hadoop since we managed simple data structures.

The DistributedCache is used to share across nodes basic information about the dataset, such as the number of rows, the number of columns, which column is the target feature. In mRMR it is also used to share what features are selected after iterations.

[1]https://github.com/nophiq/mahout

Finally we provide a complexity analysis of the implemented algorithms. Here we take into account the algorithm only and not the complexity introduced by Apache Hadoop. However, in Chapter 5 we show the Apache Hadoop overhead in managing the entire architecture, which affects the overall performance.

3.3.1 Mutual information Implementation

Java libraries for mutual information, such as JavaMI[2], require the full array of observations of both variables as input. This interface is not affordable in Big Data scenario where the observation array could not fit in the main memory.

For the thesis, we developed an implementation for mutual information measure for MapReduce. The first step was to store observations in a custom data structure, which logically is a dynamic co-occurence matrix. It increases both in row and column sizes as soon as new values are given. The second step was to implement a Java class able to calculate the mutual information given that custom data structure. This implementation works with discrete values only and Appendix B reports the code in Java.

3.3.2 Ranking implementation

Ranking calculates a coefficient, based on the Pearson correlation, between each feature and the target one. The best features are the ones with highest coefficients.

In this algorithm, we required column-normalized data as input, which means each feature vector is modeled as a gaussian distribution with zero mean and variance equals to one. In such situation, the Pearson coefficient between two features x_i and y_i with n samples is given by the Formula 3.9.

$$r_{x,y} = \frac{1}{n}\sum_{i=1}^{n} x_i y_i \tag{3.9}$$

This is a nice property for a MapReduce job, the final $r_{x,y}$ value is calculated in the reduce step, but partial sums can be locally determined with the combiner, significantly reducing the amount of data moved through the network. The ranking coefficient based on the Pearson correlation estimation is shown in Formula 3.10.

$$\text{coeff} = -\frac{1}{2}\log\left(1 - r_{x,y}^2\right) \tag{3.10}$$

[2]http://www.cs.man.ac.uk/ pococka4/JavaMI.html

The MapReduce job has the pseudo code in Listing 3.1, 3.2 and it consists of two steps:

- **Map step** (parallel over records): iterate over the training records. For each feature x an intermediate key-value object is outputted where the key is the feature index and the value is the product of the feature and target values.
- **Reduce step** (parallel over features): for each candidate feature the reducer collects data to calculate the ranking coefficient. It basically sums up all the value and output the final value for ranking, Formula 3.10.

Listing 3.1: Map step of ranking algorithm in MapReduce

```
FOR each $x \in$ feature set:
  $i$ <- feature index
  $v$ <- target feature value
  value <- $v * x$
  output($i$, value)
```

Listing 3.2: Reduce step of ranking algorithm in MapReduce

```
// The reducer input is composed by a key $i$ and a list
// of intermediate objects, referenced as "objects" in
// the pseudo code
sum <- 0
FOR each (value) $\in$ objects:
  sum <- sum + value

$r$ <- sum / n
coeff <- $-0.5 * \log\left(1 - r^2\right)$
output($i$, coeff)
```

In the implementation the combiner is used to aggregate locally the data outputted by the mappers. Finally, a post-processing step is required in order to sort the result and eventually filter it by the number of required feature.

Complexity analysis

For the complexity analysis we define first the variables used. The dataset has m features (the target feature is not taken into account here) and n rows or instaces. The number of nodes in the cluster is b.

Ranking algorithm is composed by one MapReduce job only. Afterwards a post-process step is required to sort the ranking in descending order. We describe the MapReduce job complexity analysis:

- **Mapper**: mappers read the dataset by rows only once. For each row and for each feature, mappers output an intermediate object. This task can be run in parallel across nodes. The complexity of the mapper is as follows:
$$O\left(\frac{n \cdot m}{b}\right) \tag{3.11}$$
- **Reducer**: all reducers read mappers output. For each feature reducers output a record with its ranking value. So the final output has n records. The complexity of the reducer is as follows:
$$O\left(\frac{n \cdot m}{b}\right) \tag{3.12}$$

The workload of the first MapReduce job (both mappers and reducers) is evenly distributed across nodes, so the algorithm is not affected by the skewness of data, i.e. a non uniform distribution in a dataset.

The complexity of the entire job is the sum of the mapper and reducer steps:

$$O\left(\frac{n \cdot m}{b} + \frac{n \cdot m}{b}\right) = O\left(\frac{n \cdot m}{b}\right) \tag{3.13}$$

The performance time should increase as the dataset grows and decrease as more parallelism is introduced in the system. In Chapter 5 we show results and comments.

3.3.3 mRMR implementation

In this section we introduce how mRMR algorithm has been implemented in MapReduce fashion, from high level point of view up to the Java code reported in the Appendix C.

mRMR is an iterative and greedy algorithm, which means for ten selected features there are ten iterations and once a feature has been selected it will not be removed later on. Each iteration aims to select one feature among all the *candidate features*, i.e. those not yet selected, by means of two MapReduce jobs. The first one calculates the mutual information for each candidate feature, while the second one simply selects the best feature according to the highest value of mutual information.

Each record of the training dataset is a set $(\bar{x}_i, c_i)$, where $\bar{x}_i$ is the input feature vector and c_i is the target class. The input vector is partitioned in candidate and selected features, labeled respectively as $\bar{x}_{i_c}$ and $\bar{x}_{i_s}$ ($\bar{x}_{i_c} \cup \bar{x}_{i_s} = \bar{x}_i$). At the first iteration there is no feature selected, $\bar{x}_{i_s} = \emptyset$ and $\bar{x}_{i_c} = \bar{x}_i$.

Given this description, the first MapReduce job has the pseudo code in Listing 3.3, 3.4 and it consists of two steps:

- **Map step** (parallel over records): iterate over the training records $(\bar{x}_{i_c}, \bar{x}_{i_s}, c_i)$. For each candidate feature $x_k \in \bar{x}_{i_c}$ and for each selected feature $x_j \in \bar{x}_{i_s}$ an intermediate key-value object is outputted where the key is k and the value is a tuple of value of features k and j. Furthermore, for each x_k the mapper outputs one more object, where key is k and value is a tuple composed by feature value and target value.

- **Reduce step** (parallel over features): for each candidate feature the reducer collects data to calculate the mutual information and mRMR value through the custom data structure explained in Section 3.3.1.

Listing 3.3: Map step of mRMR algorithm in MapReduce

```
// The mapper input is $(\bar{x}_{i_c}, \bar{x}_{i_s}, c_i)$
FOR each $x_k \in \bar{x}_{i_c}$:
  $v$ <- $c_i$
  output($k$, ($x_k$, $v$))

  FOR each $x_j \in \bar{x}_{i_s}$:
    $v$ <- $x_j$
    output($k$, ($x_k$, $v$))
```

Listing 3.4: Reduce step of mRMR algorithm in MapReduce

```
// The reducer input is composed by a key $k$ and a list
// of intermediate objects, referenced as "objects" in
// the pseudo code.
FOR each ($x_k$, $v$) $\in$ objects:
  if $v$ is target_class:
    // store into the custom data structure
    storeTarget($x_k$, $v$, target)
  else
    index = <index_of_feature_v>
    // store into the custom data structure
    storeFeature($x_k$, $v$, index)

// calculate mRMR of the candidate feature
mrmr <- retrieveMRMR()
output($k$, mrmr)
```

storeTarget and *storeFeature* are functions able to store data into dynamic co-occurence matrix (Section 3.3.1), from which *retrieveMRMR* calculate the mRMR of the candidate feature.

Complexity Analysis

For the complexity analysis we define first the variables used. The dataset has m features (the target feature is not taken into account here) and n rows or instaces. s is the number of feature to select and b is the number of nodes in the cluster.

mRMR is an iterative algorithm and at each iteration two MapReduce jobs are executed. Therefore we show the complexity of each mapper and reducer. Considering the i-th iteration we have:

- **First MapReduce job, mapper**: mappers read the dataset by rows only once. For each row and for each candidate feature, mappers output intermediate objects once with the target and for each selected feature values. This task can be run in parallel across nodes. The complexity of the mapper is as follows:

$$O\left(\frac{n \cdot (m-i+1) \cdot i}{b}\right) \tag{3.14}$$

- **First MapReduce job, reducer**: all reducers read mappers output. For each candidate feature reducers output a record with its mRMR value. So the final output has $m - i + 1$ records. The complexity of the reducer is as follows:

$$O\left(\frac{n \cdot (m - i + 1) \cdot i}{b}\right) \tag{3.15}$$

- **Second MapReduce job, mapper**: there is only one map running that copies and paste the input to the output. The complexity is the following:

$$O\left(m - i + 1\right) \tag{3.16}$$

- **Second MapReduce job, reducer**: only one reducer is running which scans all the mRMR values given as input and outputting the highest one. The complexity is:

$$O\left(m - i + 1\right) \tag{3.17}$$

The workload of the first MapReduce job (both mappers and reducers) is evenly distributed across nodes, so the algorithm is not affected by the skewness of data, i.e. a non uniform distribution in a dataset.

What affects the algorithm performance is the first MapReduce job, which is run s times. Therefore, the complexity of the entire job is the following:

$$\begin{aligned}
& O\left(\frac{1}{b}\sum_{i=1}^{s}\left[n \cdot (m - i + 1) \cdot i + n \cdot (m - i + 1) \cdot i\right]\right) \\
= \; & O\left(2\frac{1}{b}\sum_{i=1}^{s}\left[n \cdot (m - i + 1) \cdot i\right]\right) \\
= \; & O\left(2\frac{1}{b}\sum_{i=1}^{s}\left[n \cdot m \cdot i - n \cdot i^2 + n \cdot i\right]\right) \\
= \; & O\left(2\frac{1}{b}\left[n \cdot m\frac{s\,(s+1)}{2} - n\frac{s\,(s+1)\,(2s+1)}{6} + n\frac{s\,(s+1)}{2}\right]\right) \\
= \; & O\left(\frac{n \cdot m \cdot s^2}{b}\right)
\end{aligned}$$

In the last transformation, we considered that s is always less or equal to m by definition. For this reason, the first term is leading the complexity.

We expect an increasing performance time as either the dataset or the number of feature to select grows. Introducing more parallelism in the system the time should decrease. In Chapter 5 we show results and comments.

Chapter 4

Deployment

Amazon provides a user friendly service to run MapReduce jobs on their cloud infrastructure. As for every service its costs depends on the quality and quantity of the required resources. Machines with lots of resources are more expensive than the basic ones and short term jobs are cheaper and the long term ones.

Some professors and researchers at Université Libre de Bruxelles are already facing Big Data issues in their research. Their number is intended to increase in the next few years and rely on third-party cloud infrastructure could be too expensive. On the other hand an high performing computing center (HPC) is available for university members.

For these reasons, we developed an Apache Hadoop user friendly service on top of the HPC to meet researcher needs, similar to Amazon EMR with simpler features.

4.1 Amazon Elastic MapReduce

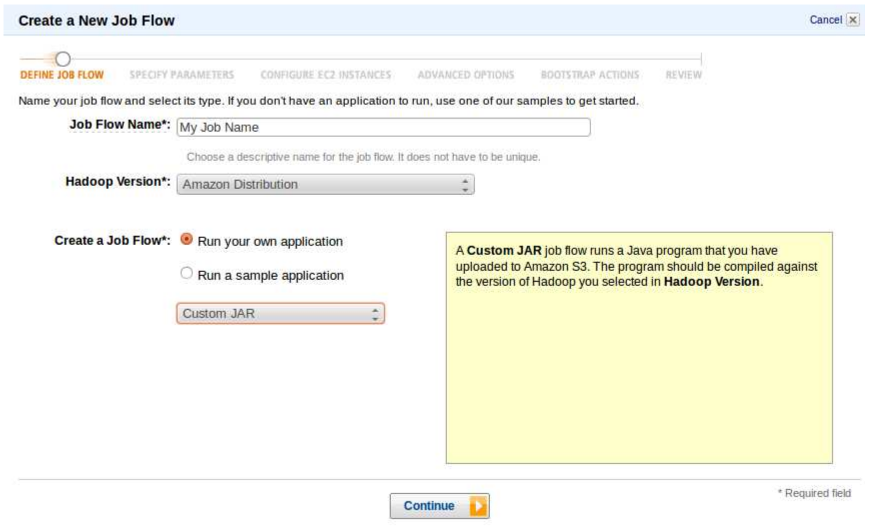
The first deployments were done on Amazon Elastic MapReduce (Amazon EMR)[1] because it provides Apache Hadoop as a service and we focused on the algorithm implementation only. Through a friendly user interface and a guided setup it is possible to easily configure cluster size, node types and submit custom jobs.

Amazon EMR is a webservice built on top of Amazon EC2[2], which offers scalable virtual machines, and Amazon S3[3], designed to be the storage system for cloud applications. Every time a new job is committed, the

[1]http://aws.amazon.com/elasticmapreduce/

[2]http://aws.amazon.com/ec2/

[3]http://aws.amazon.com/s3/


Create a New Job Flow

Cancel

DEFINE JOB FLOW · SPECIFY PARAMETERS · CONFIGURE EC2 INSTANCES · ADVANCED OPTIONS · BOOTSTRAP ACTIONS · REVIEW

Name your job flow and select its type. If you don't have an application to run, use one of our samples to get started.

Job Flow Name*: My Job Name

Choose a descriptive name for the job flow. It does not have to be unique.

Hadoop Version*: Amazon Distribution

Create a Job Flow*: [✓ Run your own application] Run a sample application

Custom JAR

A **Custom JAR** job flow runs a Java program that you have uploaded to Amazon S3. The program should be compiled against the version of Hadoop you selected in **Hadoop Version**.

* Required field

Continue

Figure 4.1: Amazon EMR, step 1: defining job name, Apache Hadoop version and submitted job

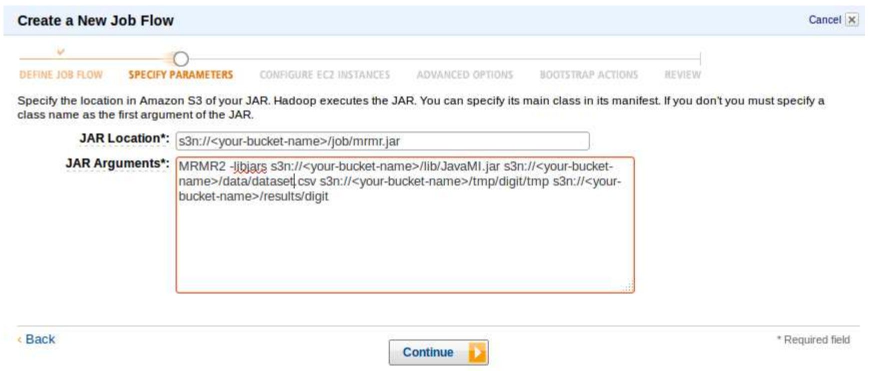
system launches as many virtual machines as specified nodes, prepares the environment with Apache Hadoop and moves the original data from S3 storage to the HDFS. When the job finishes, the output is copied in the S3 storage for later analysis.

We review the six-steps process for submitting an Apache Hadoop job on Amazon EMR in the following list:

1. **Define job flow** (Figure 4.1): each submitted job on Amazon is called *job flow* and it is identified by a name. In this step the user selects which version of Apache Hadoop to use, between the original version[4] and the one provided by the software company MapR[5]. Finally it is possible to submit five different kind of jobs: Apache Hive (Section 2.4.2), Apache Hadoop Streaming (Section 2.3.10), Apache Pig[6], Apache HBase (Section 2.4.1) and a custom job compressed as jar file.

2. **Parameters** (Figure 4.2): the user specifies where is the job and what are the related parameters. Every time there is a reference to a filesystem location, the user must refer to the S3 storage ("s3n://" prefix). The Amazon system automatically converts the S3 prefix in the HDFS one when data moves to or from the local HDFS.

[4]https://hadoop.apache.org/
[5]http://www.mapr.com/
[6]http://pig.apache.org/



Specify the location in Amazon S3 of your JAR. Hadoop executes the JAR. You can specify its main class in its manifest. If you don't you must specify a class name as the first argument of the JAR.

JAR Location*: s3n://<your-bucket-name>/job/mrmr.jar

JAR Arguments*: MRMR2 -libjars s3n://<your-bucket-name>/lib/JavaMI.jar s3n://<your-bucket-name>/data/dataset.csv s3n://<your-bucket-name>/tmp/digit/tmp s3n://<your-bucket-name>/results/digit

Figure 4.2: Amazon EMR, step 2: defining job parameters

3. **Configure EC2 instances** (Figure 4.3): in Amazon an *instance* is a virtual machine running on the cloud infrastructure. In this step the user defines the kind of instances for both master and slaves. Several instance types are possible[7] depending on the job requirements.

4. **Advanced options**: among the advanced options available it is useful to setup the log path. Through the command line is possible to submit jobs and track its progress status. In Amazon EMR the user can redirect the output to an external file identified in the S3 storage system.

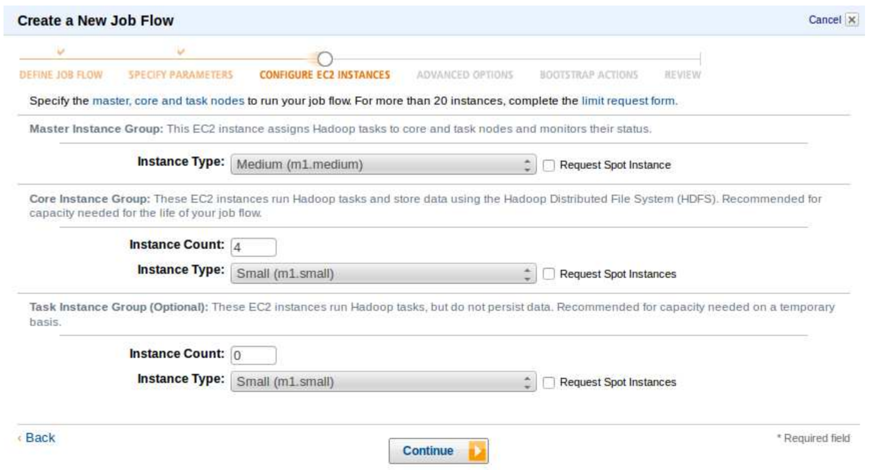
5. **Bootstrap**: in this step the user can customize Apache Hadoop or the global environment before the job is actually submitted in the cluster.

6. **Review**: an overview of the parameters set before confirming the job flow.

4.2 Hydra, the computing center

Hydra[8] is the dedicated high performing computing center at Université

[7]http://aws.amazon.com/elasticmapreduce/#instance
[8]https://cc.ulb.ac.be/hydra/



Specify the master, core and task nodes to run your job flow. For more than 20 instances, complete the limit request form.

Master Instance Group: This EC2 instance assigns Hadoop tasks to core and task nodes and monitors their status.

Instance Type: Medium (m1.medium) ☐ Request Spot Instance

Core Instance Group: These EC2 instances run Hadoop tasks and store data using the Hadoop Distributed File System (HDFS). Recommended for capacity needed for the life of your job flow.

Instance Count: 4
Instance Type: Small (m1.small) ☐ Request Spot Instances

Task Instance Group (Optional): These EC2 instances run Hadoop tasks, but do not persist data. Recommended for capacity needed on a temporary basis.

Instance Count: 0
Instance Type: Small (m1.small) ☐ Request Spot Instances

Figure 4.3: Amazon EMR, step 3: defining instance type for both master and slaves

Libre de Bruxelles[9] (ULB) and Vrije Universiteit Brussel (VUB)[10]. It is also in the context of the Vlaams Super Computer[11] and the Consortium des Equipement pour le Calcul Intensif[12].

4.2.1 Hydra as batch system

Hydra is designed as a batch system, in which the distributed resources (storage systems, networks, software licences) are abstracted as a single entity. The concept behind this approach is related to the cluster utilization: resources are better exploited if the system is considered as one entity rather than a set of computers. Batch systems have four different components:

- **Master node**: the master node manages the resources and jobs in the cluster. One node is usually dedicated to this task only.
- **Submit or interactive nodes**: these nodes are the entry point for users to manage their workloads. From here they can submit and monitor jobs.
- **Computer nodes**: computer nodes are the workhorses of the system. Their role is to execute submitted jobs and to communicate with the

[9]http://www.ulb.ac.be/
[10]http://www.vub.ac.be/
[11]https://vscentrum.be/
[12]http://hpc.montefiore.ulg.ac.be/

master node.

- **Resources**: it is the collection of resources available in the compute nodes, such as networks, storage systems, license managers and so forth.

Hydra is a balanced cluster, i.e. suitable for batch, multi-cores (SMP) and multi-nodes (MPI) jobs[13]. Each node runs CentOS 6.4 operating system with Linux version 2.6.32-279.el6.x86_64 and it is connected into the cluster with 1Gb ethernet and 10Gb Infiniband networks[14]. The total disk space available to store the jobs output is 5.2TB. The Java version used to run Apache Hadoop jobs is 1.6.0_27.

There are two master nodes running several services such as the MOAB cluster management system. Their specifications are the following:

- 2x AMD Opteron 6100 2.3GHz with 8 cores each.
- 16GB DDR3 RAM 1.333Ghz.
- 2x 146GB HDD SAS 10K RPM in RAID1.
- 1x Gb Ethernet connection.
- 1x HP 4X QDR Infiniband connection on a PCI express 2x card.
- 2x power supply units with UPS protection.

In the system, 64 computer nodes execute submitted jobs by the users with a total of 1024 cores:

- 2x AMD Opteron 6100 2.3GHz with 8 cores each.
- 64GB DDR3 RAM 1.333GHz.
- 2 x 500GB SATA 7.2k in RAID0 with a 1GB RAM controller.
- 1x Gb Etherner connection.
- 1x HP 4X QDR Infiniband connection on a PCI express 2x card.
- Redundant power supply units with UPS protection.

4.2.2 User jobs in Hydra

Users access to Hydra through ssh[15] with their credentials provided by the computing center. All actions are executed through the shell. Many standard general purpose applications are directly available in the working environment, while custom or specific tool versions can be plugged in as modules.

[13]https://cc.ulb.ac.be/hydra/documentation.php
[14]http://en.wikipedia.org/wiki/InfiniBand
[15]http://en.wikipedia.org/wiki/Secure_Shell

Every job submitted by users is defined by the programs to be executed, the resources required for the execution, the environment in which the programs are executed and the credentials associated with the user and the job. This information is handled by two core softwares on Hydra: Torque and Moab (Section 4.2.3).

The life cycle of a job can be divided into four stages:

1. **Creation**: a script is used to specify all the information related to required resources and the instructions to execute. Some resource parameters include the job name, how long a job should run (*walltime*) and how many cores are necessary.

2. **Submission**: once a job is submitted, the policies set dictates its priority.

3. **Execution**: when the job gets the resource required it starts its execution and the user can monitor its status.

4. **Finalization**: at the end of the execution stdout and stderr (see Glossary) streams are copied to the user working directory.

Further constraints or specifications about resources can be collected through *queues*. Some queues are already defined in Hydra for instance to limit the execution to a compute node subsets.

4.2.3 Torque and Moab

Torque [55] is a resource manager software that provides control over batch jobs and distributed computing resources[16]. It is an open source project able to handle large clusters with tens of thousands of nodes and jobs. It provides a fault tolerant system and can be integrated with Moab workload manager[17] to improve overall utilization, scheduling and administration on a cluster.

Torque, like other resource managers, provides low-level functionality to start, hold, cancel, and monitor jobs. Thereby there are three main tasks, and relative command line tools, in Torque:

- **Job submission**: it is accomplished using the *qsub* command, which takes job information and instructions through a dedicate file or stdin

[16]http://www.adaptivecomputing.com/products/open-source/torque/

[17]http://www.adaptivecomputing.com/products/hpc-products/moab-hpc-basic-edition/

(see Glossary). If the user wants to submit the same job a large number of times, Torque provides multiple job submission functionality. In the system, each executing job is uniquely identified.

- **Monitoring jobs**: Torque allows users to monitor submitted jobs with *qstat* command.

- **Cancelling jobs**: With *qdel* command, Torque will kill the running processes. The only parameter accepted is the ID of an executing job.

At a high level, Moab [12] applies policies and extensive optimizations to jobs, services, and other workload across the cluster. It increases the system resources availability providing a fault tolerant system, a diagnostic service and visualization tools for statistics, reporting and charts.

As job scheduler, Moab dynamically matches jobs to nodes to better take advantage of cluster resources. Achieving fairness and optimal usage of compute resources is complex and the scheduler has to to determine when, where, and how to run jobs to optimize the cluster. Scheduling decisions can be categorized as follows [12]:

- **Traffic control**: the scheduler must prevent different jobs contending for the same resources, otherwise cluster performance decreases and jobs might even fail. Tracking which resources are assigned to jobs handles this issue.

- **Policies**: it defines rules regarding who is able to use what resources.

- **Optimizations**: when resource demand exceeds supply, intelligent scheduling decisions facilitate job completions and reduce queue time.

Moab can optimize different workload types within the same cluster simultaneously. In the following we introduce the four workload types managed by Moab [12]:

- **Batch workload**: required resources and execution environment for the job are described in a file. With a batch job, the job is submitted to a job queue, and it is run somewhere on the cluster as resources become available.

- **Interactive Workload**: interactive workloads provide immediate response to submitted instructions. These jobs are typically routed into the cluster via a web or other graphical terminal.

- **Calendar workload**: this type of workload executes the submitted job at a particular time and eventually at a regular basis. Here time contraints can be rigids or flexibles.
- **Service workload**: Moab can also respond to externally generated request for service running inside the cluster. For example, a third-party service can query data in a parallel database deployed and running in the cluster, Moab executes the request and returns the response.

For the purpose of our master thesis all our workloads for performance tests belonged to the batch type.

4.2.4 Deploying Apache Hadoop on Hydra

Hydra computing center did not have Apache Hadoop installed in the cluster. These are the challenges we faced in order to successfully deploy Apache Hadoop on Hydra:

- **Credentials**: we had simple user privileges only. We were not able to change directly cluster or node configurations according to our needs.
- **Working directory**: We needed to install Apache Hadoop in a local directory of the node. In Hydra architecture all directories are shared but the temporary one, located at "/tmp", which is automatically cleaned up every 24h. It was then not possible run long-term jobs more than one day. As shown in Chapter 5 some tests required much longer running time.
- **Assigning jobs to disjoint nodes**: in Torque it is not possible to submit two different jobs to two disjoint node sets, limiting the deploying flexibility of our contributions (explained in Section 4.3).
- **Testing**: We run our jobs on a shared environment, which means there were other jobs competing with us for the resources and this slowed down testing. Furthermore, the only way to login on Hydra is through ssh, making impossible accessing the Apache Hadoop web interface (Section 2.2.4) to check job status online.
- **Hydra downtime**: scheduled or unexpected downtime let Hydra not available for testing.
- **Concurrent jobs**: Apache Hadoop allows multiple jobs running on the same cluster, but it is not possible to run two different installations of the software in the nodes. Here the problem is related to the

communication between master and slaves, because messages are sent through some ports that identify the service on a host and two different services cannot rely on the same port to communicate. Changing the port number for each deployment is not good solution and it does not scale well.

One final challenge was the several Apache Hadoop releases published during master thesis work. For every new release some maintenance was required. The first version adopted was 0.20 and it was a thoughtful choice because the software supports two API interfaces, the deprecated one and the current one. This choice let the entire literature and code available be useful in our first approach into Apache Hadoop.

Amazon Elastic MapReduce supports Apache Hadoop version 1.0.3. To avoid compatibility issues between the two different software versions we adapted our MapReduce algorithm for the cloud infrastructure.

Finally, the Apache Hadoop version deployed on Hydra was 1.1.2 (released in February 2013[18]), which fixed some major bugs about long-term jobs, the ones we were interested in to run to understand the scalability of our algorithms.

4.3 Contributions

In this section we present our contributions in deploying Apache Hadoop on a computing cluster. We developed a user-friendly service which dynamically deploys Apache Hadoop on cluster of different sizes and runs the job submitted by the user.

The registered user on Hydra interacts with the service by command line and a set of script files. We designed the system based on how Amazon Elastic MapReduce works, i.e. the user setup the most important parameters and the service manages the rest. The number of required nodes and the job to submit is the only information to provide. Apache Hadoop installation, configuration and job submission is automatically managed by the service. In Sections 4.3.2 and 4.3.1 we describe in detail how this service works and how we overcame the limitation reported in Section 4.2.4.

The service is composed of a set of scripts, grouped in two categories. One is meant to deploy Apache Hadoop on the available resources (Section 4.3.2) and the other is customizable by the user to provide job information (Section 4.3.1).

[18]https://hadoop.apache.org/releases.html

4.3.1 Job submission

Every user that wants to access the service through Hydra has to customize two scripts.

The most important script is the submission script, which is the one submitted to the system through the *qsub* command (Section 4.2.3). It is in charge of executing several tasks:

- Setting torque parameters, such as the walltime and the number of nodes for the cluster.
- Retrieving information about the assigned resources, such as node identifiers. Every nodes has an unique host identifier, composed by a fixed prefix string and a number, examples are *nic42* and *nic64*. In configuring the cluster architecture we decided to assign the master role to the node which has the lowest number, the slave role to all the others. Since the maximum cluster size used during the test was 8, we decided to setup the master also as a slave.
- Connecting to each node by ssh to run Apache Hadoop automatic deployment (Section 4.3.2).
- Start the cluster on the master node, executing the job script and stop the cluster (Section 4.3.2).

The job script configures the environment variables and contains all the information and instructions to execute the job itself.

4.3.2 Automatic deployment

Each node receives some parameters related to the cluster configuration, such as the role (master or slave), the working directory and job id. With this information it runs two scripts in sequence:

- **Installation**: the installation script takes care to clean up the environment if some old hanging job are still running purposeless, eventually freeing ports necessary for the communication between master and slaves. Finally it installs Apache Hadoop on a dedicated local node directory. Since "/tmp" did not fit our needs, we asked the computing center support to create a directory specifically for Apache Hadoop ("/hadoop").
- **Configuration**: based on the node roles, the script properly configures Apache Hadoop configuration files.

The master node only executes two other scripts: once the configuration is done it starts the cluster, when the submitted job is completed it stops the cluster and clean up the environment.

4.3.3 Considerations

Since we had simple user privileges only, we could not deployed a stable and always running Apache Hadoop on Hydra. The solution we adopted allows to run Apache Hadoop clusters on several and disjoint subsets of nodes. If nodes of two different deployments overlap then both system will crash. To run tests in parallel, we setup three fixed disjoint cluster with the collaboration of the computing center support: with 4, 6 and 8 nodes, each one identified by a queue. We could have run tests sequentially, but with this approach the testing phase were faster. Fixing the nodes increases the queue time for the jobs, since resources on exactly those nodes have to be available, but actually it did not affect significantly tests.

Though the scripts were tested on Hydra only, they rely on an environment mainly based on open source software: the operating system in each node is Linux and the combination of Torque and Moab is a popular among HPC. So we are confident that, eventually with small changes, the scripts could run in other cluster centers.

4.3.4 How to use the service

In this section we describe how to use the service on Hydra. We designed the service to have a simple interface, such as the Amazon Elastic MapReduce one, in which the user sets the most important information about the cluster and the job.

In this example we suppose the user has already logged in the system through its credentials and it wants to submit a job in a 4 nodes Apache Hadoop deployment. As explained in Section 4.3.1 and 4.3.2 the first file the user has to modify is the submission script, whose excerpt is reported in Listing 4.1.

Listing 4.1: Excerpt of the submission script, where Torque parameters are set

```
#PBS -o out
#PBS -e err
#PBS -N hadoop4
#PBS -l mem=19gb
#PBS -l nodes=4:ppn=1
#PBS -l walltime=1:00:00

JOB_FULLPATH=/<path>/<to>/job_script
```

In the script we defined the filename of the output and error log file, respectively *out* and *err*; the job name, *hadoop4*; the main memory for each node, 19GB; the walltime for the script. The number of nodes and the number of cores per node are fixed, because of the constraints imposed on the resources. For a complete documentation about Torque parameters see [55].

In Listing 4.2 we report an excerpt of the job script. The user has to first load the dataset into HDFS, execute the job and finally copy the results from HDFS to a local directory. In this example we are running mRMR algorithm over *dataset.csv* input, which has 10^7 rows and 100 features. At the end of the job 5 features are selected and the job result is stored in *output* directory. [33, 60, 50] report a full description of Apache Hadoop and Apache Mahout command line tools and parameters.

Listing 4.2: Excerpt of the job script, for the job submission

```
${HADOOP_CMD}/hadoop fs -put
    /<local>/<path>/<to>/dataset.csv dataset.csv

${MAHOUT_HOME}/bin/mahout hadoop jar
    /<path>/<to>/<lib>/mahout_feature.jar
    org.apache.mahout.feature.mrmr.MRMRDriver -i
    dataset.csv -o output -t 1 -nr 10000000 -nc 100 -nf 5

${HADOOP_CMD}/hadoop fs -get output
    /<local>/<path>/<to>/output
```

Finally, from Hydra command line the user submits the job on the system through the qsub command (Section 4.2.3). Without any options in qsub, Apache Hadoop can be deployed on every node, but for our performance tests we needed to run three clusters in parallel, where each node set is disjointed from the others. This constraint is introduced by using three different queues: *hadoop4*, *hadoop6*, *hadoop8*. They specify the cluster with 4, 6 and 8 nodes respectively.

In our example we submit the job to the cluster with four nodes (Listing 4.3).

Listing 4.3: Submitting job to Hydra

```
$ qsub -q hadoop4 subsmission_script
```

Through qstat command (Section 4.2.3) the user can check the status of the job on Hydra.

Chapter 5

Performance tests and results

In Big Data scenario the dataset has a very large number of instances and features. A properly designed algorithm for MapReduce runs in a reasonable time if there is a fair amount of computational resources in the cluster.

This chapter shows that our implemented algorithms scale in Apache Hadoop context. We were interested in understanding the level of parallelism and scalability of our feature selection algorithms and the overhead introduced by Apache Hadoop in managing jobs and cluster.

5.1 Datasets

We built three groups of artificial datasets in order to run tests with different input size.

The first group is a set of six datasets, all of them with the same number of columns: 10^2 features and 1 target feature. Each one has a different number of rows: 10^1, 10^2, 10^3, 10^4, 10^5, 10^7. Globally, the dataset sizes vary from about 2KB to 2GB. Values are discrete and each feature has three possible alternatives: -2, 0 and 2, while the target feature is either 0 or 1. The dataset structure has been taken from bioinformatics, where typically each feature expresses a gene regulation and the target feature values represent different experiment conditions. Some real datasets with the same structure are available online[1].

The R code generating the data is reported in Listing 5.1. In line 10, we decided to set the vector of probability weights for obtaining the sampled elements to better represent the real sparsity of bioinformatics datasets.

[1]http://penglab.janelia.org/proj/mRMR/

Listing 5.1: R code to generate discrete values. Each feature has three possible values: -2, 0 and 2, while the target feature is a binary value: 0, 1.

```r
library(MASS)

nRows <- <NROWS>
nFeatures <- <NCOLS>
out <- file(<FILENAME>, "w")
outputRows <- <OUTPUT_NROWS_AT_A_TIME>

classValues <- c(0,1)
featureValues <- c(-2,0,2)
featureProb <- c(0.15,0.7,0.15)

loops <- as.integer(nRows/outputRows)

for(i in 1:loops) {
  matrix <- sample(classValues, outputRows, replace=T)
  for (i in 1:nFeatures) {
    matrix <- c(matrix, sample(featureValues, outputRows,
        replace=T, prob=featureProb))
  }

  dim(matrix) <- c(outputRows, nFeatures+1)

  write.table(matrix, file=out, sep=",",
      col.names=FALSE, row.names=FALSE)
}
close(out)
```

The second group is a set of three datasets of continuous values. They have 99 features and 1 target feature. The number of rows is 10^5, 10^6 and 10^7. Since these datasets are supposed to run with the Ranking algorithm, each feature has to be sampled from a normal distribution with mean equals to zero and variance to one. The R code generating the data is reported in Listing 5.2.

Listing 5.2: R code to generate continuous values. Each feature (column) has to be sample from a normal distribution with mean equal to zero and variance to one.

```r
nrows <- <NROWS>
ncols <- <NCOLS>
filename <- <FILENAME>

alldata <- c()

for (i in 1:ncols) {
  alldata <- c(alldata, rnorm(n=nrows, m=0, sd=1))
}
d <- matrix(alldata, nrow=nrows, ncol=ncols)
write.table(d, file=filename, sep=",", row.names=FALSE,
    col.names=FALSE)
```

The third group is a set of three datasets of discrete values. Here the number of rows is fixed at 10^2 and the number of columns is 10^3, 10^4 and 10^5. Values are arranged as the first group and the R code generating the dataset is reported in Listing 5.1, with parameters properly set.

5.2 Tests and considerations

MapReduce is a well-accepted framework for data intensive applications over computing clusters. Researches related to Apache Hadoop, benchmarking or scalability of the paradigm focused on three categories: testing the software over different infrastructures [7], modeling MapReduce performance [38] or studying the scalabity of new algorithms [54, 58, 40, 44, 59]. Our work belongs to the last category.

We first verified each algorithm works accurately and then we ran extensive tests to figure out the scalability and parallelism of MapReduce feature selection algorithms. Further tests were run to understand the overhead Apache Hadoop introduces in running jobs.

For mRMR, from authors website[2] tools and some small datasets are available. For each dataset the output of our algorithm and the one provided by the authors matches. As described in Section 3.3.3, at each iteration the best feature is selected through two MapReduce jobs, the first assigns the mRMR value to each candidate feature and the second filters the best feature. In Section 5.3 we discuss the performance of each job.

[2]http://penglab.janelia.org/proj/mRMR/

For Ranking, the output of our distributed algorithm in MapReduce matches the one from an R script which executes the same job. In Section 5.5 we discuss the performance of the MapReduce job.

In parallelism tests (Section 5.3.1 and 5.5.1), we proved that there is a real benefit in increasing the number of working units in executing jobs. In all clusters we forced first the algorithm to run with one map task only, which means there is one node working and the job is executed sequentially. The performance is compared with the one in which Apache Hadoop automatically decides the number of map tasks, i.e. the level of parallelism.

In overhead tests (Section 5.3.2) we were interested in showing that Apache Hadoop spends time in managing and setting jobs and tasks. This overhead is significant when the MapReduce job runs over a small dataset or Apache Hadoop parameters are not well set.

There were some issues in forcing the number of map tasks in jobs. *mapred.map.tasks* is a parameter that could be set by command line or the Java code, but it is actually not taken into account from Apache Hadoop. We solved this problem using NLineInputFormat (Section 2.3.8), setting the number of lines per map task equals both to the number of rows and to 1, for parallelism and overhead tests respectively.

5.2.1 Apache Hadoop setup

We ran several tests in clusters with 4, 6 and 8 nodes. For each node, one core was reserved for the jobs. We used Apache Hadoop 1.1.2 and the default configurations excepts for the following:

- *dfs.replication*: we set the block replication parameter at 2. Each block is stored twice in the HDFS.
- *mapred.child.ulimit*: we set the maximum virtual memory, in KB, of a process launched by the MapReduce framework at 16777216.
- *mapred.child.java.opts*: we set the Java opts for the task tracker child processes at -Xmx512m

We changed the replication parameter to decrease the load time of the dataset, while we fixed memory limits related to the job to better control the environment under which the MapReduce jobs were executed.

5.2.2 Retrieving information

Apache Hadoop logs provide data about the number of records and bytes in input and output in mappers, reducers and eventually combiners. Furthermore it is reported the number of map and reduce tasks, information about the memory used and other deatils.

What is missing is the amount of time spent in executing the map and reduce steps, the ones we were interested in for performance tests. These quantities can be obtained indirectly from the job progress logs. Thereby, we developed for our tests a custom tool that parsed the job logs to retrieve execution time information.

In all figures related to mRMR, but Figure 5.5, the vertical axis represents the execution time to select the i-th feature only and not the time required to select up to the i-th feature. The latter can be derived by summing up the time of current and previous iterations.

5.3 mRMR MapReduce job

5.3.1 Parallelism tests

In Figure 5.1, we report performance tests to prove that increasing the level of parallelism reduces the running time in Apache Hadoop context, as expected from the complexity analysis in Section 3.3.3. This concept is very important for the purpose of the thesis, because it proves that when the feature selection algorithm is fed with a Big Data dataset it is possible to add more nodes (or cores) to run the job in a reasonable time.

The difference between the two settings increases as the algorithm selects more features because the workload is higher. We show and discuss in other tests (Section 5.3.3) that this benefit increases as the dataset size grows. For this reason we show here tests with the largest dataset, while the full range of tests is reported in Figure E.1, E.2 and E.3.

5.3.2 Overhead tests

In Figure 5.2, we report performance tests to show the overhead introduced by Apache Hadoop in each MapReduce job. Every time a map task starts, the Java Virtual Machine (JVM) takes few seconds to set the environment up. The more map tasks are executed the more is the overhead. The overhead is clear even with small datasets. Figure 5.2 shows tests with dataset of 10^4 rows, the full range of tests is reported in Figure E.4, E.5 and E.6.

There are several parameters the user can set[3][4][5] and in this section

[3]http://hadoop.apache.org/docs/r1.1.2/core-default.html
[4]http://hadoop.apache.org/docs/r1.1.2/hdfs-default.html
[5]http://hadoop.apache.org/docs/r1.1.2/mapred-default.html

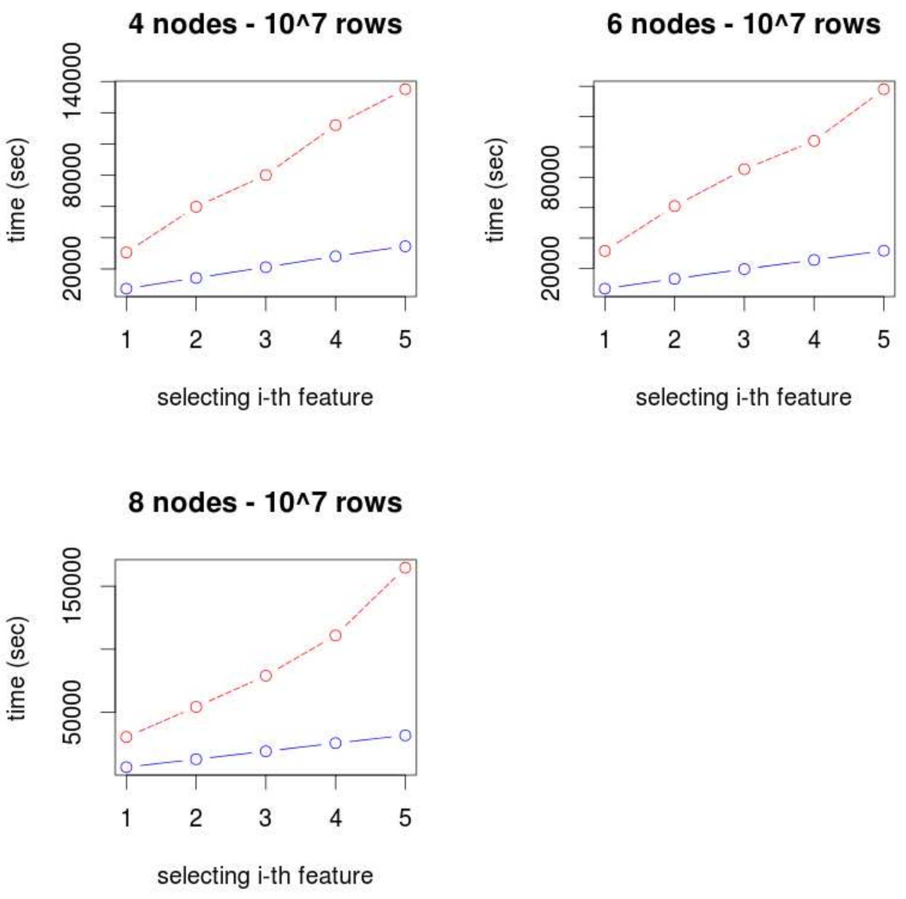


Figure 5.1: mRMR: parallelism tests in each cluster using the dataset of 10^7 rows. The red dashed line represents tests with one map task running (sequential execution), while the blue solid one the number of map tasks is decided by Apache Hadoop (parallel execution).

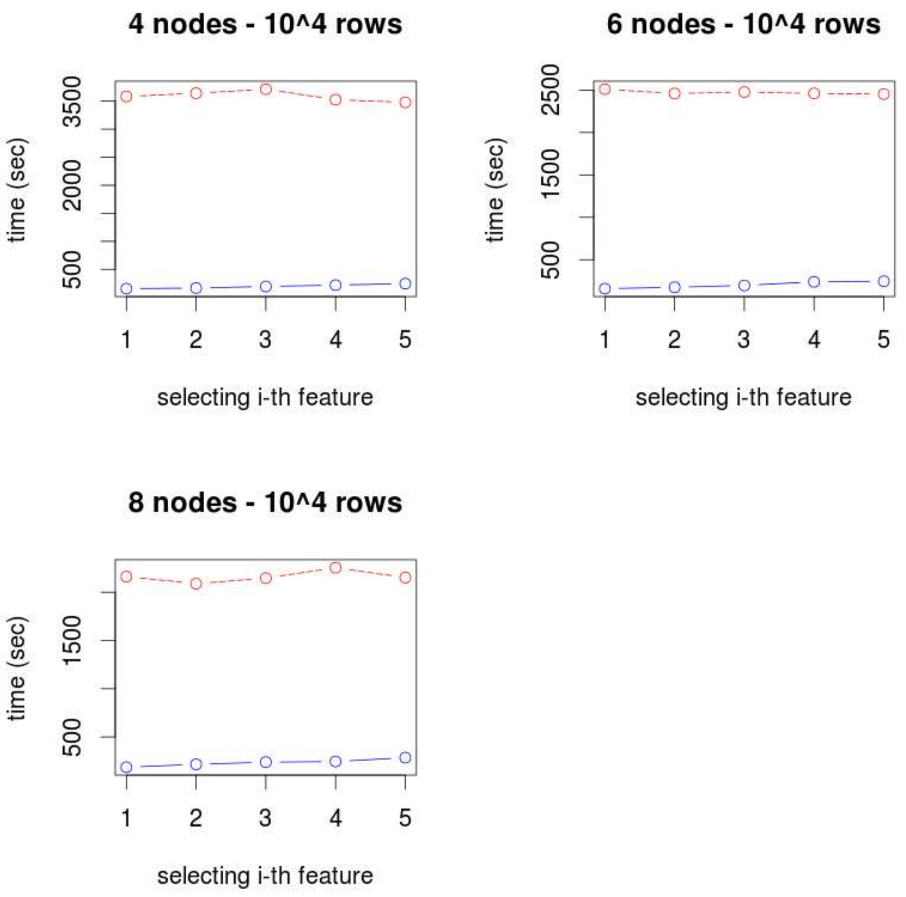


Figure 5.2: mRMR: overhead tests in each cluster using the dataset of 10^4 rows. The red dashed line represents tests with the maximum number of map task running, while the blue solid one the number of map tasks is decided by Apache Hadoop.

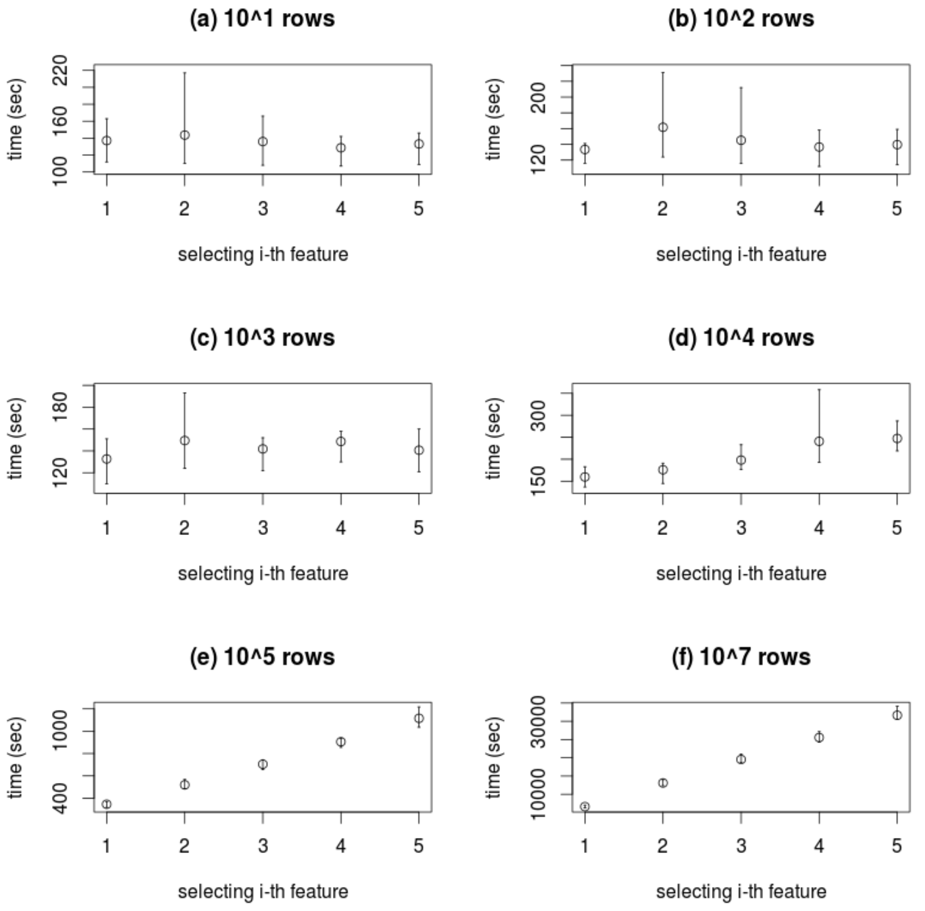


Figure 5.3: mRMR: scalability tests with 6 nodes. The number of map tasks is set by Apache Hadoop.

we showed that even one of them badly set could really affect the overall performance. Finding the best tuning for all parameters is an important step in running MapReduce jobs on Apache Hadoop.

5.3.3 Scalability tests

In Figure 5.3 we show the results of scalability tests, where the number of map tasks is set up by Apache Hadoop. Since we ran several tests for each situation, we show with an empty dot the average of the results and with a bar the range of values (from minimum to maximum). Tests report that as the dataset grows the execution time increases, agreeing with our analysis (Section 3.3.3).

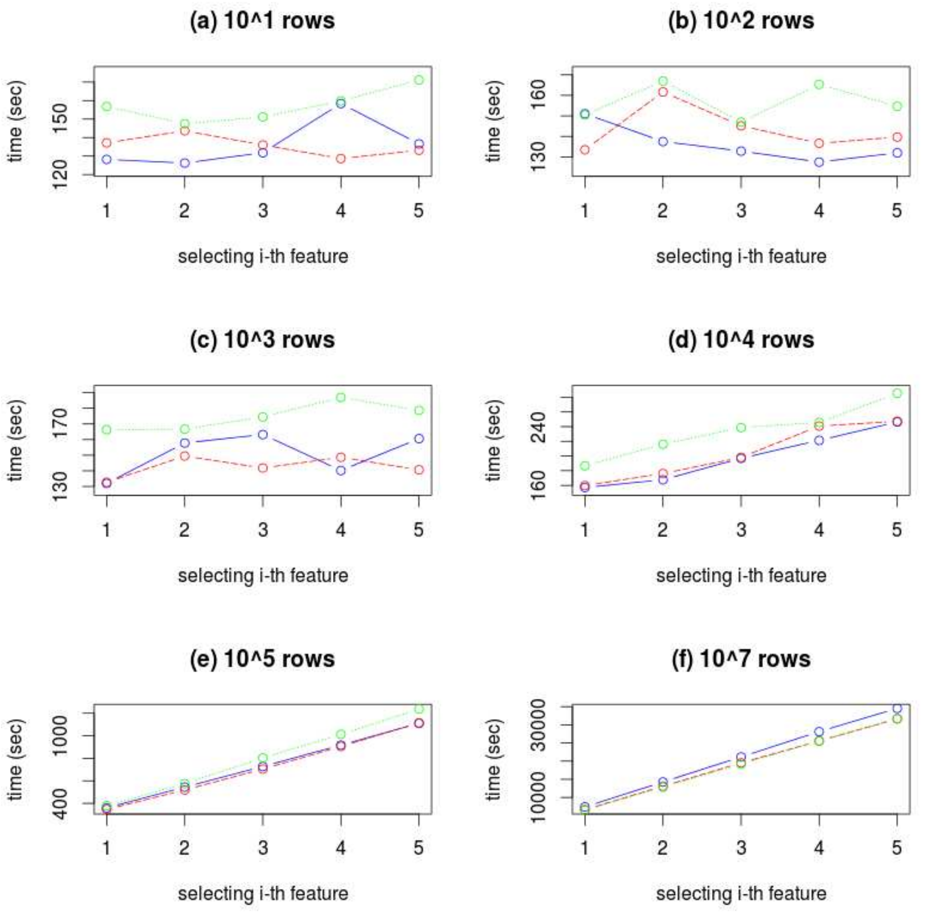


Figure 5.4: mRMR: comparing performance time across cluster size. The blue solid line is Apache Hadoop with 4 nodes, the red dashed line with 6 nodes and the green dotted line with 8 nodes.

Furthermore, in these plots we can observe that with few instances (less than 10^4 rows) selecting the 5th feature almost takes as much as time as selecting the 1st one, because the time overhead related to the JVM and managing the cluster heavily bias the performance. Here we provide tests in cluster with 6 nodes, we had similar results in the other clusters (see Figure E.7 and E.8).

In Figure 5.4 we compare performance time over different cluster sizes. We expected the time decreases as the cluster size increases, but results show differently. In Figure 5.4 (a), (b), (c) and (d) the 8 nodes cluster performs worse than the others, because of Apache Hadoop overhead in managing a bigger cluster with small data. The most valuable observations come

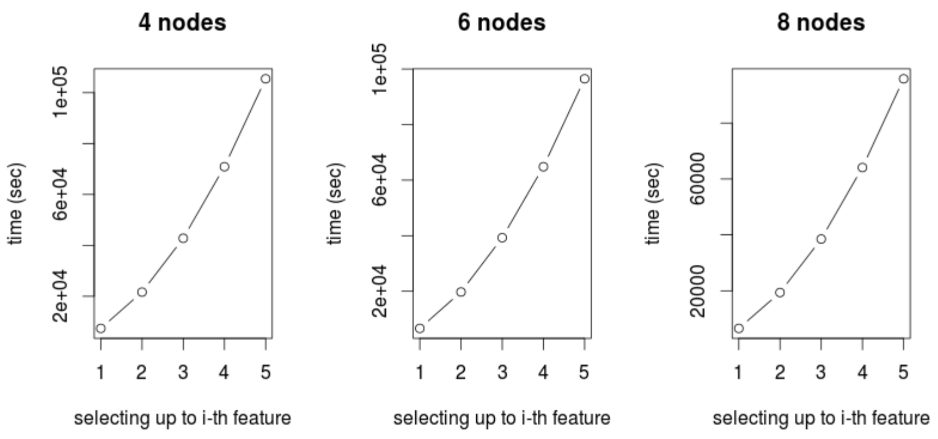


Figure 5.5: mRMR: the performance time increases exponentially with respect to the number of feature to select. The dataset has 10^7 rows and 10^2 columns.

from Figure 5.4 (e) and (f): even with 2GB dataset all clusters performed almost the same. As explained in [54], it is because the dataset is still small and a more suitable one should be at least one hundred times more bigger. Unfortunately, with the resources available (8 nodes with one core per node only), scalability tests with that dataset is unfeasible. In [58] the cluster relied on 416 core distributed in 52 nodes.

In Figure 5.5 we show the time required to execute mRMR up to the i-th in all clusters. According to our complexity analysis (Section 3.3.3) the performance time is exponentially proportional to the number of feature to select.

Finally, we fed the algorithm with datasets where the number of rows was fixed and the number of columns increased. As expected from the complexity analysis (Section 3.3.3), the time increases as the number of dataset columns grows, Figure 5.6.

5.4 Best feature MapReduce job

In mRMR the output of the first MapReduce job is a set of key-value records, where the key is the candidate feature index and the value is its mRMR. The second MapReduce job filters this set to get the best candidate and, according to the theoretical complexity analysis provided in Section 3.3.3, the performance time should increase with the number of columns, but our tests over datasets with 10^4 and 10^5 columns do not agree with our expectations, Figure 5.7. As already described in Section 5.3.3 that is related

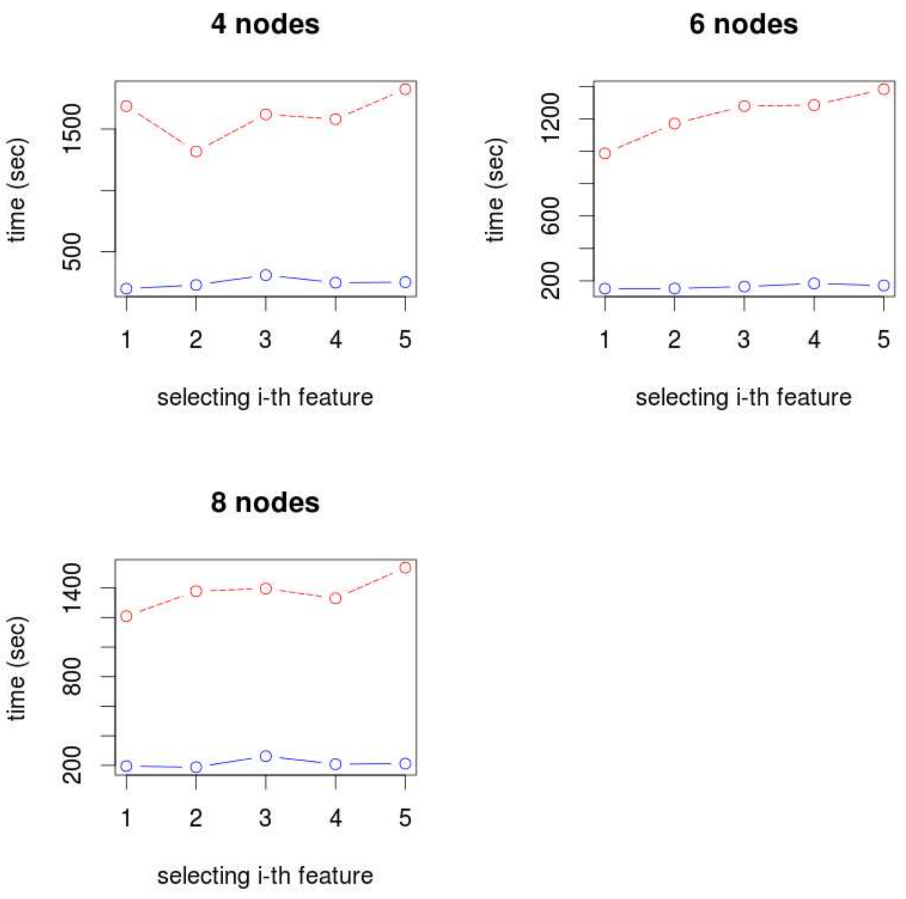


Figure 5.6: mRMR: scalability tests as the dataset size increases by the number of features. The number of rows is fixed at 10^2. The blue solid line represents a dataset with 10^3 features, while the red dashed line 10^4 features.

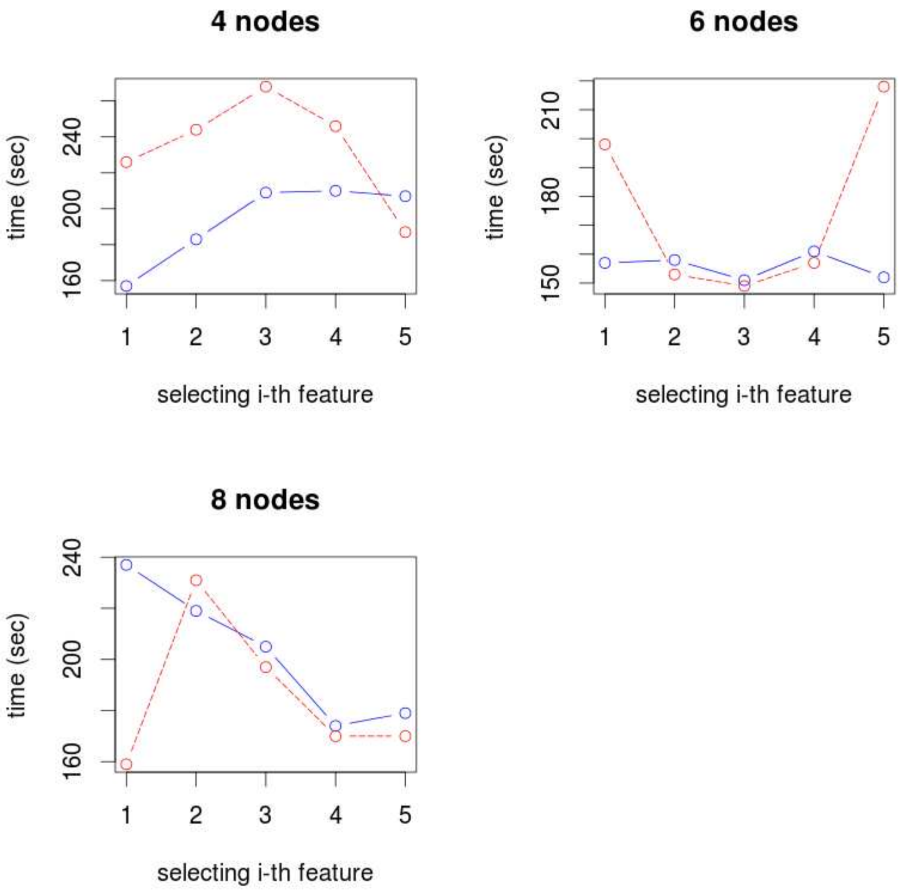


Figure 5.7: Best feature: scalability tests as the dataset size increases by the number of features. The number of rows is fixed at 10^2. The blue solid line represents a dataset with 10^3 features, while the red dashed line 10^4 features.

to the dataset size (small) and unfortunately it was not possible to run the algorithm over bigger datasets because not feasible with the resources available.

Comparing the execution times of both MapReduce jobs, the best feature is much faster than the mRMR. However, in MapReduce the former still takes around 2 or 3 minutes to complete, which is a lot for its kind of job. In a sequential way it is likely to perform better even in a real context scenario, but in Apache Hadoop it is not possible to execute some jobs in MapReduce fashion and others in sequential way.

In Figure E.9, E.10 and E.11 we ran the algorithm with different datasets where the number of columns were fixed and the number of rows varied. As

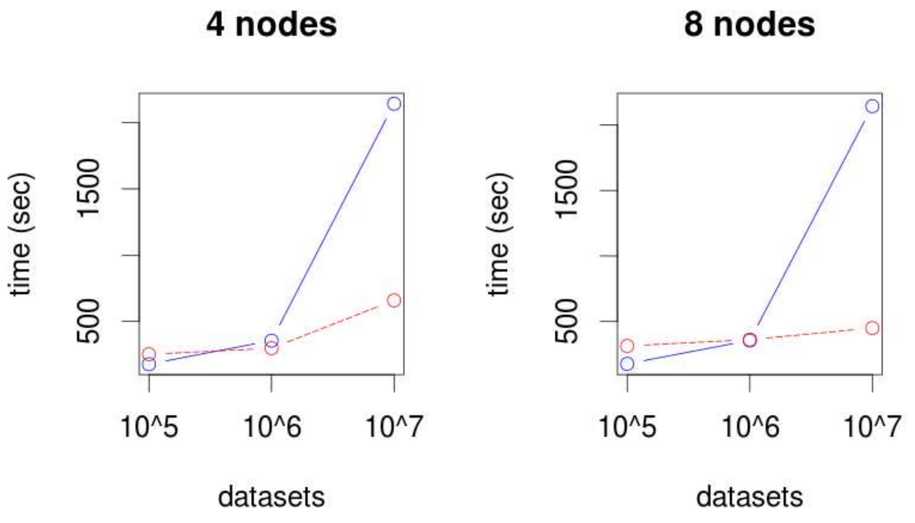


Figure 5.8: Ranking: parallelism tests in 4 and 8 clusters. The blu line represents tests with one map task running (sequential execution), while the red dashed one the number of map tasks is decided by Apache Hadoop (parallel execution).

expected, performance tests are not significantly different, since the execution time of the algorithm depends on the number of features and not on the number of rows.

5.5 Ranking MapReduce job

In this section we show the performance tests ran with Ranking. We proceeded assessing the level of parallelism and the scalability of the algorithm, leaving out overhead tests since it has been strongly shown in tests for mRMR.

5.5.1 Parallelism tests

The results shown in Figure 5.8 agree with those reported in mRMR. Increasing the level of parallelism reduces the running time, as expected from the complexity analysis in 3.3.2.

5.5.2 Scalability tests

Results related to scalability agree with our expectations, when the dataset used is the largest one. In Figure 5.9 it is shown that increasing the number

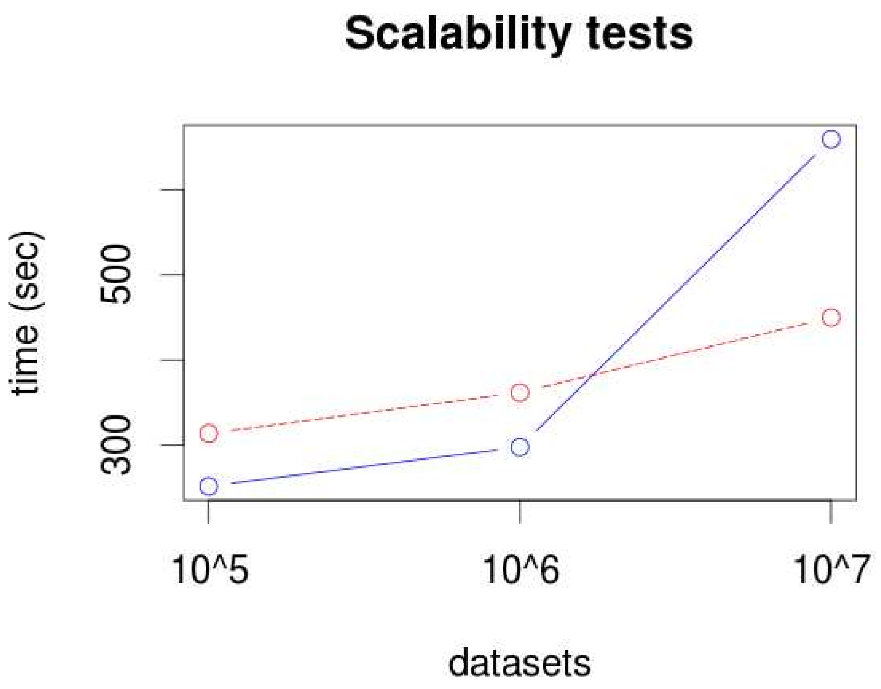


Figure 5.9: Ranking: scalability tests in cluster with 4 and 8 nodes, the blue solid and red dashed lines respectively.

of nodes in the cluster decreases the execution time, and in Figure 5.10 the number of features affects the performance.

From these tests we see the two algorithms fit differently the MapReduce paradigm. In Ranking the Apache Hadoop overhead does not affect the execution time as much as in mRMR. Furthermore, the workload of Ranking and mRMR at first iteration is similar, but the time differs of about one order of magnitude (about 600 seconds and 6000 seconds respectively), Figure 5.3 and 5.9. This is an insight to better investigate, but our guess is related to the use of combiner in Ranking, which reduces the amount of data moved through the network and read by reducers.

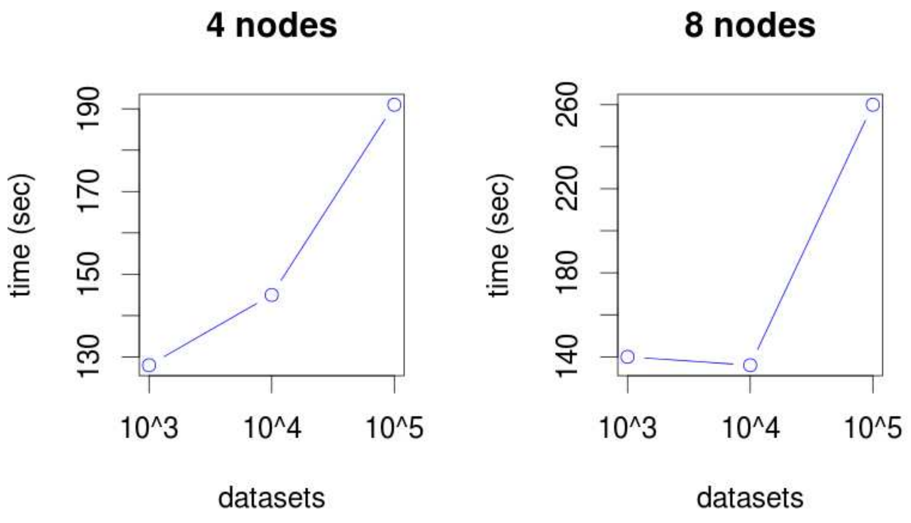


Figure 5.10: Ranking: scalability tests as the dataset size increases by the number of features. The number of rows is fixed at 10^2. The datasets used had 10^3, 10^4 and 10^5 features.

Chapter 6

Future works and conclusions

In this document we reviewed limitations that Database Management Systems have in handling very large dataset. They were the solution usually adopted as data storage system, but their archicture does not scale well in a distributed system because complexity and constraints are introduced. With the coming of Big Data a paradigm change was required and the new way in storing and querying data emerged to tackle these limits and the applications were grouped under the term NoSQL.

Google contributed in overcoming DBMS issues publishing two papers, the Google file system [21] and MapReduce [14], which are cornerstones in the new way of storing and querying data. Apache Hadoop is the result of the open source community in letting this technology available for everyone and a whole new ecosystem of closed and open sourced software is built on top of it for different goals.

We were interested in how to design and implement machine learning algorithms in MapReduce, because in domains in which Big Data is common such as internet algorithms, computational biology or social link analysis, traditional single machine algorithms may no longer be feasible to produce models in reasonable time. Our focus was on feature selection algorithms because they provide a way to reduce noise and remove irrelevant features from the dataset, leading to a better generalization of prediction models.

For these reasons we considered feature selection algorithms in MapReduce for distributed systems, which are not available in popular open source tools such as Apache Mahout. We designed and implemented two feature selection algorithms: mRMR and Ranking and we released the code as a public Java library for Apache Mahout available online. At the ULB computing center we developed an user friendly service, which dynamically deploy Apache Hadoop on the nodes and runs MapReduce jobs on top of it. Though the service ran on very limited computational resources with respect to those used in related works, our extensive performance tests let us understand the scalability of our algorithms and the overhead introduced by Apache Hadoop in managing jobs and the cluster. Furthermore, working with small datasets showed insignificant insights for our main goals, results with larger input only matched our scalability expectations.

There are different directions to go further in this work. In this document we took into account mRMR and Ranking algorithms, but there are other feature selection algorithms to explore, implement and compare in MapReduce; all researchers and enterprises will benefit in enriching the Java library with those algorithms. Furthermore each implemented algorithm is the result of some design choices and the implementation affects the performance.

Our performance analysis were limited to changing few parameters only, a broader analysis can be done taking into account all the possible settings of Apache Hadoop, the cluster and nodes. This research will detailed explore what is the best configuration for the given algorithm and dataset, eventually predicting ahead the amount of execution time. The model can also define the computational resources required by the job and estimate its cost, providing useful insights and information.

Finally, we were first in deploying Apache Hadoop on the computing center. The computational resources assigned to our tests were not always enough to show our theoretical analysis. Next step is demanding more resources to run more extensive tests and deploying Apache Hadoop on a dedicated environment.

Glossary

- **Checksum**: it is a small piece of content computed from any digital data to detect errors. When an user retrieves a piece of content from a service it receives the related checksum. Locally, the user computes the checksum and compares it with the one got from the service, if they do not match some errors occured during the transmission.
- **Denormalization**: it is the process through which data becomes redundant in the database to improve the performance of the system.
- **Horizontal scaling** or **scale out**: it usually refers to tying multiple independent computers together to provide more processing power. In Relational DBMS it splits one or more tables by row within a single instance of a schema.
- **Schema**: a database schema defines the structure of a database in a formal language.
- **Sharding**: sharding is a concept related to horizontal scaling, but the partition is across multiple instances of the schema, improving the performance at read time.
- **Stdin, stdout, stderr**: they are standard streams through which a computer program is connected while it executes. The first is data going into a program, the second is the sink for output data and the last one collects error messages or diagnostics.

List of variables

Here we list the variables used throughout the document:

- n: number of records or instances of the dataset.
- m: number of features or columns of the dataset.
- c: it defines the target class.
- S: set of feature selected.
- s: number of feature to select.
- b: number of nodes in the cluster.

Bibliography

[1] Alekh Agarwal, Olivier Chapelle, Miroslav Dudḱ, and John Langford. A reliable effective terascale linear learning system. 2011.

[2] Leonard E. Baum, Ted Petrie, George Soules, and Norman Weiss. A maximization technique occurring in the statistical analysis of probabilistic functions of markov chains. *The Annals of Mathematical Statistics*, 41(1):pp. 164–171, 1970.

[3] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, March 2003.

[4] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[5] Sarah Boslaugh and Paul Andrew Watters. *Statistics in a nutshell - a desktop quick reference*. O'Reilly, 2008.

[6] Andrei Z. Broder. On the resemblance and containment of documents. In *In Compression and Complexity of Sequences (SEQUENCES'97*, pages 21–29. IEEE Computer Society, 1997.

[7] Jeff Buell. A benchmarking case study of virtualized hadoop performance on vmware vsphere 5. Technical report, VMware, Oct 2011.

[8] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. Technical report, Google Inc., 2006.

[9] Chakra Chennubhotla and Allan D. Jepson. Half-lives of eigenflows for spectral clustering. pages 689–696, 2002.

[10] Cheng T. Chu, Sang K. Kim, Yi A. Lin, Yuanyuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learn-

ing on multicore. In Bernhard Schölkopf, John C. Platt, and Thomas Hoffman, editors, *NIPS*, pages 281–288. MIT Press, 2006.

[11] Dorin Comaniciu and Peter Meer. Mean shift: A robust approach toward feature space analysis. pages 603–619, 2002.

[12] Adaptive Computing. Moab workload manager. `http://docs.adaptivecomputing.com/mwm/mwmAdminGuide-7.2.3.pdf`.

[13] Corinna Cortes and Vladimir Vapnik. Support-vector networks. In *Machine Learning*, pages 273–297, 1995.

[14] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. Technical report, Google Inc., 2004.

[15] Pierre A. Devijver and Josef Kittler. *Pattern Recognition: A Statistical Approach.* Prentice Hall, January 1982.

[16] Michel Dubois and Christoph Scheurich. Memory access dependencies in shared-memory multiprocessors. *IEEE Trans. Softw. Eng.*, 16(6):660–673, June 1990.

[17] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory access buffering in multiprocessors. In *25 years of the international symposia on Computer architecture (selected papers)*, ISCA '98, pages 320–328, 1998.

[18] Stéphane Gançarski, Claudia León, Hubert Naacke, Marta Rukoz, and Pablo Santini. Integrity constraint checking in distributed nested transactions over a database cluster. *CLEI Electron. J.*, 9(2), 2006.

[19] Kourosh Gharachorloo and et al. Performance evaluation of memory consistency models for shared-memory multiprocessors.

[20] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *In Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, 1990.

[21] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. Technical report, Google Inc., 2003.

[22] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent available partition-tolerant web services. In *In ACM SIGACT News*, 2002.

[23] Dan Gillick, Arlo Faria, and John Denero. Mapreduce: Distributed computing for machine learning, 2006.

[24] James R. Goodman. Cache consistency and sequential consistency, 1989.

[25] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *J. Mach. Learn. Res.*, 3:1157–1182, March 2003.

[26] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, December 1983.

[27] Mark A. Hall. Correlation-based feature selection for machine learning. Technical report, 1998.

[28] Jiawei Han, Micheline Kamber, and Jian Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2011.

[29] R. R. Hocking. The analysis and selection of variables in linear regression, 1976.

[30] Aapo Hyvarinen and Erkki Oja. Independent component analysis: algorithms and applications. *Neural Networks*, 13:411–430, 2000.

[31] Ron Kohavi and George H. John. Wrappers for feature subset selection. *Artif. Intell.*, 97(1-2):273–324, 1997.

[32] Tim Kraska, Ameet Talwalkar, John C. Duchi, Rean Griffith, Michael J. Franklin, and Michael I. Jordan. Mlbase: A distributed machine-learning system. In *CIDR*, 2013.

[33] Chuck Lam. *Hadoop in Action*. Manning, December 2010.

[34] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[35] Douglas Laney. 3D data management: Controlling data volume, velocity, and variety. Technical report, February 2001.

[36] Wentian Li. Mutual information functions versus correlation functions. *Journal of Statistical Physics*, 60:823–837, 1990.

[37] Jimmy Lin and Chris Dyer. Data-intensive text processing with mapreduce. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Companion Volume: Tutorial Abstracts*, NAACL-Tutorials '09, 2009.

[38] Xuelian Lin, Zide Meng, Chuan Xu, and Meng Wang. A practical performance model for hadoop mapreduce. In *CLUSTER Workshops*, pages 231–239, 2012.

[39] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new framework for parallel machine learning. 2010.

[40] Wei Lu, Yanyan Shen, Su Chen, and Beng Chin Ooi. Efficient processing of k nearest neighbor joins using mapreduce. *Proc. VLDB Endow.*, 5(10):1016–1027, June 2012.

[41] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press, 1967.

[42] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 135–146. ACM, 2010.

[43] Andrew McCallum, Kamal Nigam, and Lyle H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '00, pages 169–178, 2000.

[44] Ahmed Metwally and Christos Faloutsos. V-smart-join: a scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *Proc. VLDB Endow.*, 5(8):704–715, April 2012.

[45] Miller and Alan. *Subset Selection in Regression.* Chapman and Hall, 1990.

[46] Thomas M. Mitchell. *Machine Learning.* McGraw-Hill, Inc., 1997.

[47] Luis Carlos Molina, Lluís Belanche, and Àngela Nebot. Feature selection algorithms: A survey and experimental evaluation. In *Proceedings of the 2002 IEEE International Conference on Data Mining*, ICDM '02. IEEE Computer Society, 2002.

[48] Andrew Y. Ng, Michael I. Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS*, pages 849–856. MIT Press, 2001.

[49] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads programming*. O'Reilly & Associates, Inc., 1996.

[50] Sean Owen, Robin Anil, Ted Dunning, and Ellen Friedman. *Mahout in Action*. Manning, October 2011.

[51] Hanchuan Peng, Fuhui Long, and Chris Ding. Feature selection based on mutual information: criteria of max-dependency, max-relevance, and min-redundancy. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(8), 2005.

[52] Saeed K. Rahimi and Frank S. Haug. *Distributed Database Management Systems: A Practical Approach*. Wiley, August 2010.

[53] Sangwon Seo, Edward J. Yoon, Jaehong Kim, Seongwook Jin, Jin-Soo Kim, and Seungryoul Maeng. Hama: An efficient matrix computation with the mapreduce framework. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, pages 721–726. IEEE Computer Society, 2010.

[54] Sameer Singh, Jeremy Kubica, Scott Larsen, and Daria Sorokina. Parallel large scale feature selection for logistic regression.

[55] Garrick Staples. Torque resource manager. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06. ACM, 2006.

[56] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel R. Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alexander Rasin, Nga Tran, and Stan B. Zdonik. C-store: A column-oriented dbms. In *VLDB*, 2005.

[57] Yee Whye Teh, David Newman, and Max Welling. A collapsed variational bayesian inference algorithm for latent dirichlet allocation. 2006.

[58] Abhishek Verma, Xavier Llorà, David E. Goldberg, and Roy H. Campbell. Scaling genetic algorithms using mapreduce. In *Proceedings of the 2009 Ninth International Conference on Intelligent Systems Design and Applications*, ISDA '09, pages 13–18. IEEE Computer Society, 2009.

[59] Guozhang Wang, Marcos Vaz Salles, Benjamin Sowell, Xun Wang, Tuan Cao, Alan Demers, Johannes Gehrke, and Walker White. Behavioral simulations in mapreduce. *Proc. VLDB Endow.*, 3(1-2):952–963, September 2010.

[60] Tom White. *Hadoop: The Definitive Guide.* O'Reilly Media, Inc., 2009.

[61] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition (Morgan Kaufmann Series in Data Management Systems).* Morgan Kaufmann Publishers Inc., 2005.

[62] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, 2012.

[63] Hongyuan Zha, Xiaofeng He, Chris Ding, Horst Simon, and Ming Gu. Spectral relaxation for k-means clustering. pages 1057–1064. MIT Press, 2001.

[64] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-scale parallel collaborative filtering for the netflix prize. In *Proc. 4th Int'l Conf. Algorithmic Aspects in Information and Management, LNCS 5034*, pages 337–348. Springer, 2008.

Appendix A

Apache Mahout tools

This appendix describes by categories all the available tools in Apache Mahout.

A.1 Preprocessing

arff.vector converts ARFF (Attribute-Relation File Format) file in vectorized format. It describes a list of instances sharing a set of attributes and it is widely used in Weka software[1] for data mining.

seq2encoded generates encoded sparse vector from text sequence files.

seq2sparse generates sparse vector from text sequence files. This command converts SequenceFile intermediate data into vectorized data. This is basically one of the most used tool available to handle data and it's usually run after a command that converts your raw data into SequenceFile intermediate data. For text-based data it automatically does the following actions: compute dictionary, compute feature weights and create vector for each document using word-integer mapping and feature-weight.

seqdirectory generate sequence files of text from a directory. Since it doesn't run in MapReduce it cannot handle big single dataset. A patch has been suggested[2] and integrated in future release.

seqmailarchives creates SequenceFile from a directory containing gzipped mail archives.

seqwiki creates SequenceFile from Wikipedia xml dump.

[1]http://www.cs.waikato.ac.nz/ml/weka/
[2]https://issues.apache.org/jira/browse/MAHOUT-833

A.2 Algorithms

baumwelch refers to Baum-Welch algorithm for unsupervised HMM training [2].

canopy execute canopy cluster algorithm [43], a technique for clustering large, high-dimensional datasets regardless the number of cluster of interest. To reduce the computational cost, the idea is to use a fast approximate distance metric which outputs overlapping subsets of data, called canopies. Then clustering is performed by measuring exact distances only between points that occur in a common canopy.

cleansvd verifies the **svd** command output checking related eigenvectors and eigenvalues.

cvb applies latent Dirichlet allocation [3] via Collapsed Variation Bayes [57]. It is a generative probabilistic model for collections of discrete data such as text corpora. The implemented algorithm in Apache Mahout is a variational algorithm which models the dependence of the parameters on the latent variables in an exact fashion, assuming mutually independency of latent variables.

cvb0_local run **cvb** in local memory.

dirichlet performs Bayesian mixture modeling. The idea is to use a probabilistic mixture of a number of models that explains some observed data. The non-parametric nature of this model makes it a candidate for clustering problems where the distinct number of clusters is unknown beforehand.

eigencuts performs eigencuts spectral clustering [9].

evaluateFactorization computes RMSE and MAE of a rating matrix factorization against probes.

fkemans performs fuzzy k-means, an extension of K-Means. While k-means discovers hard clusters (a point belong to only one cluster), Fuzzy K-Means discovers soft clusters where a particular point can belong to more than one cluster with certain probability.

fpg performs frequent itemset for association rules. It uses an efficient compressed dataset representation, which retains the itemset association information, from which it mines association rules.

hmmpredict generates random sequence of observations by given HMM (Hidden Markov Model). Unfortunately, there is no exhaustive documentation regarding this tool.

itemsimilarity computes the item-item-similarities for item-based collaborative filtering.

kmeans performs k-means clustering technique [41], which partitions the dataset in k cluster and each data belongs to one cluster only.

meanshift performs mean shift clustering [11], is a nonparametric clustering technique which does not require prior knowledge of the number of clusters, and does not constrain the shape of the clusters.

minhash performs k-means clustering technique [6]. Minhash clustering performs probabilistic dimension reduction of high dimensional data. The essence of the technique is to hash each item using multiple independent hash functions such that the probability of collision of similar items is higher. Multiple such hash tables can then be constructed to answer near neighbor types of queries efficiently.

parallelALS performs Alternating Least Squares with Weighted λ Regularization (ALS-WR) [64], a parallel algorithm for collaborative filtering, designed for the Netflix Prize[3].

recommendfactorized computes recommendations using the factorization of a rating matrix.

recommenditembased computes recommendations using item-based collaborative filtering.

rowsimilarity computes the pairwise similarities of the rows of a matrix.

runAdaptiveLogistic scores new production data using a trained and validated AdaptivelogisticRegression model (see trainAdaptiveLogistic command).

runlogistic runs a logistic regression model against CSV data.

spectralkmeans performs spectral k-means clustering [63, 48].

ssvd stands for Stochastic SVD, which outputs the reduced rank Singular Value Decomposition. It uses at most 3 MapReduce sequential steps (map-only + map-reduce + 2 optional parallel map-reduce jobs) to produce reduced rank approximation of U, V and S matrices[4].

svd performs Lanczos Singular Value Decomposition.

testnb tests a Naive Bayes classifier.

[3]http://www.netflixprize.com/

[4]https://cwiki.apache.org/confluence/display/MAHOUT/Stochastic+Singular+Value+Decomposition

trainAdaptiveLogistic trains an AdaptivelogisticRegression model.

trainlogistic trains a logistic regression using stochastic gradient descent.

trainnb trains a Naive Bayes classifier.

validateAdaptiveLogistic validates an AdaptivelogisticRegression model against hold-out data set.

viterbi performs Viterbi algorithm which is known as inference algorithm (synonyms: segmentation, decoding etc) for Hidden Markov Model which finds the most likely sequence of hidden states by given sequence of observed states.

A.3 Postprocessing

clusterpp helps in executing a top down hierarchical clustering. Each cluster technique outputs a description of the points and information about the clusters. Clusterpp command riorganizes the output in order to further analyze each group of point with a cluster algorithm.

clusterdump dumps cluster output to text.

cmdump dumps confusion matrix in HTML or text formats.

matrixdump dumps matrix in CSV format.

vectordump dumps vectors from a sequence file to text.

A.4 Utilities

cat prints a file or resource as the logistic regression models would see it.

lucene.vector generates vectors from a Lucene[5] index.

matrixmult executes the product of two matrices.

regexconverter converts text files on a per line basis based on regular expressions. It is useful for converting things like log files from one format to another.

split creates training and holdout set with a random 80-20 split of the dataset.

splitDataset works as split command but here the user defines the ratio between training and probe sets.

[5]http://lucene.apache.org/core/

rowid it maps datasets in SequenceFile<Text, VectorWritable> format to {SequenceFile<IntWritable, VectorWritable>, SequenceFile<IntWritable, Text>}. This command is usually chained with the transpose job [50].

transpose takes the transpose of a matrix.

Appendix B

Mutual Information for MapReduce

This Appendix shows the implementation of mutual information for MapReduce. The mutual information is calculated in the reduce step, where each pair of observation is read one at a time and stored in a custom data structure which logically represents a dynamic co-occurence matrix, implemented as *MatrixList* class (Listing B.1). The basic component of the co-occurence matrix is *RowElement* (listing not reported) which stores how many times the pair row-column value has been observed. Finally *MutualInformation* (Listing B.2) class reads the data structure to calculate the mutual information.

Listing B.1: Dynamic co-occurence matrix implementation

```java
package org.apache.mahout.feature.common.correlation;

import org.apache.mahout.feature.common.correlation.RowElement;

import java.util.ArrayList;
import java.util.Iterator;

public class MatrixList {

        protected String name;

        protected ArrayList<RowElement> elements;
        protected ArrayList<RowElement> uniqueRows;
        protected ArrayList<RowElement> uniqueCols;

        protected long occurrences;

        public MatrixList() {
                elements = new ArrayList<RowElement>();

                uniqueRows = new ArrayList<RowElement>();
                uniqueCols = new ArrayList<RowElement>();

                this.occurrences = 0;
        }
        public String getName() {
                return this.name;
        }
        public void setName(String name) {
                this.name = name;
        }
        public void store(int a, int b) {
                this.storeElement(a, b);
                this.storeUniqueRow(a);
                this.storeUniqueCol(b);

                this.occurrences = this.occurrences + 1;
        }
        private void storeElement(int row, int col) {
                boolean isNew = true;
                for (RowElement e: elements) {
                        if (e.getRow() == row && e.getCol() ==
                            col) {
                                isNew = false;
                                e.increaseOccurrence();
                                break;
                        }
                }
                if (isNew) {
                        elements.add(new RowElement(row, col));
                }
        }
        private void storeUniqueRow(int row) {
                boolean isNew = true;
                for (RowElement e: uniqueRows) {
                        if (e.getRow() == row) {
                                isNew = false;
                                e.increaseOccurrence();
                                break;
                        }
                }
                if (isNew) {
                        uniqueRows.add(new RowElement(row, -1));
                }
        }
        private void storeUniqueCol(int col) {
                boolean isNew = true;
                for (RowElement e: uniqueCols) {
                        if (e.getCol() == col) {
                                isNew = false;
                                e.increaseOccurrence();
                                break;
                        }
                }
                if (isNew) {
                        uniqueCols.add(new RowElement(-1, col));
                }
        }
        public Iterator elementsIterator() {
                return elements.iterator();
        }
        public long getOccurrences() {
                return this.occurrences;
        }
        public long getRowOccurrences(int row) {
                for (RowElement e: uniqueRows) {
                        if (e.getRow() == row) {
                                return e.getOccurrences();
                        }
                }
                return -1;
        }
        public long getColOccurrences(int col) {
                for (RowElement e: uniqueCols) {
                        if (e.getCol() == col) {
                                return e.getOccurrences();
                        }
                }
                return -1;
        }
}
```

Listing B.2: Mutual information calculation

```
package org.apache.mahout.feature.common.correlation;

```

```java
import org.apache.mahout.feature.common.correlation.MatrixList;
import org.apache.mahout.feature.common.correlation.RowElement;

import java.util.Iterator;

public class MutualInformation {

        private int instanceNumber;

        public MutualInformation() {}

        public double computeResult(MatrixList matrix) {

                long tot = matrix.getOccurrences();
                Iterator<RowElement> iterator =
                    matrix.elementsIterator();

                double mi = 0.0;
                while (iterator.hasNext()) {
                        RowElement e = iterator.next();

                        double pxy = (double) e.getOccurrences()
                            / tot;
                        double px = (double)
                            matrix.getRowOccurrences(e.getRow()) /
                            tot;
                        double py = (double)
                            matrix.getColOccurrences(e.getCol()) /
                            tot;

                        if (pxy > 0.0) {
                                mi = mi + (pxy * (Math.log(
                                    pxy/(px*py) ) / Math.log(2)));
                                //mi = mi + (pxy * (Math.log(
                                    pxy/(px*py) )));
                        }
                }

                return mi;
        }
}
```

Appendix C

mRMR in MapReduce

This Appendix reports code excerpts of mapper (Listing C.1) and reducer (Listing C.2) of mRMR in MapReduce.

Listing C.1: mRMR mapper

```
// targetIndex is the index of the target feature
// columnNumber is the total number of columns
// selected is the set of already selected features
public void map(LongWritable index, Text record, Context
    context) throws IOException, InterruptedException {

        String[] values = record.toString().split(",");
        for (int i=0; i<columnNumber; i++) {
                // i is the index of the candidate feature
                if (selected.contains(""+i) || i == targetIndex)
                    continue;

                keyOut.set(i);
                textOut.set(values[i]+","+values[targetIndex]+",t");
                context.write(keyOut, textOut);

                for (int j=0; j<columnNumber; j++) {
                        // j is the index of the already selected
                            feature
                        if (!selected.contains(""+j)) continue;

                        keyOut.set(i);
                        textOut.set(values[i]+","+values[j]+",f,"+j);
                        context.write(keyOut, textOut);
                }
        }
}
```

Listing C.2: mRMR reducer

```
public void reduce(IntWritable index, Iterable<Text> items,
    Context context) throws IOException, InterruptedException {

        MatrixList target = new MatrixList();
        ArrayList<MatrixList> features = new
            ArrayList<MatrixList>();

        for (Text item: items) {
                String[] values = item.toString().split(",");

                int candidateValue = Integer.parseInt(values[0]);
                String type = values[2];

                if (type.equals("t")) {

                        int targetValue =
                            Integer.parseInt(values[1]);
                        target.store(candidateValue, targetValue);

                } else if (type.equals("f")) {

                        int featureValue =
                            Integer.parseInt(values[1]);
                        String featureName = values[3];

                        boolean isNew = true;
                        for (MatrixList matrix: features) {
                                if
                                    (matrix.getName().equals(featureName))
                                    {
                                        isNew = false;
                                        matrix.store(candidateValue,
                                            featureValue);
                                        break;
                                }
                        }
                        if (isNew) {
                                MatrixList matrix = new
                                    MatrixList();
                                matrix.setName(featureName);
```

```
                                    matrix.store(candidateValue,
                                        featureValue);
                                    features.add(matrix);
                            }

                    }
            }

            MutualInformation mi = new MutualInformation();

            double sum_features = 0.0;
            for (MatrixList f: features) {
                    sum_features = sum_features + mi.computeResult(f);
            }

            double sum_target = mi.computeResult(target);

            double coefficient = 1.0;
            if (features.size() > 1) coefficient = (1.0 / ((double)
                features.size()));
            double correlation = sum_target - (coefficient *
                sum_features);

            context.write(new LongWritable(0), new
                Text(index.get()+","+String.format("%.5f",
                correlation)));
    }
```

Appendix D

Ranking in MapReduce

This Appendix reports code excerpts of mapper (Listing D.1), combiner (Listing D.2) and reducer (Listing D.3) of Ranking algorithm in MapReduce.

Listing D.1: Ranking mapper

```
public void map(LongWritable index, Text record, Context
    context) throws IOException, InterruptedException {

        String[] values = record.toString().split(",");
        double targetValue =
            Double.parseDouble(values[targetIndex]);

        for (int i=0; i<columnNumber; i++) {
                if (i == targetIndex) continue;

                double candidateValue =
                    Double.parseDouble(values[i]);

                keyOut.set(i);
                valueOut.set(targetValue*candidateValue);

                context.write(keyOut, valueOut);
        }
}
```

Listing D.2: Ranking combiner

```
public void reduce(IntWritable index, Iterable<DoubleWritable>
    items, Context context) throws IOException,
    InterruptedException {

        double partial_rho = 0.0;
        for (DoubleWritable item: items) {
                partial_rho = partial_rho + item.get();
        }

        context.write(index, new DoubleWritable(partial_rho));
}
```

Listing D.3: Ranking reducer

```
public void reduce(IntWritable index, Iterable<DoubleWritable>
    items, Context context) throws IOException,
    InterruptedException {

        double rho = 0.0;
        for (DoubleWritable item: items) {
                rho = rho + item.get();
        }
        rho = rho / rowNumber;

        double mionc = - 0.5 * Math.log(1 - (rho*rho));

        context.write(index, new Text(""+mionc));
}
```

Appendix E

Tests

In this Appendix we report all performance tests run to understand several aspects of both Apache Hadoop and the feature selection algorithms implemented. Each figure provides a description of itself.

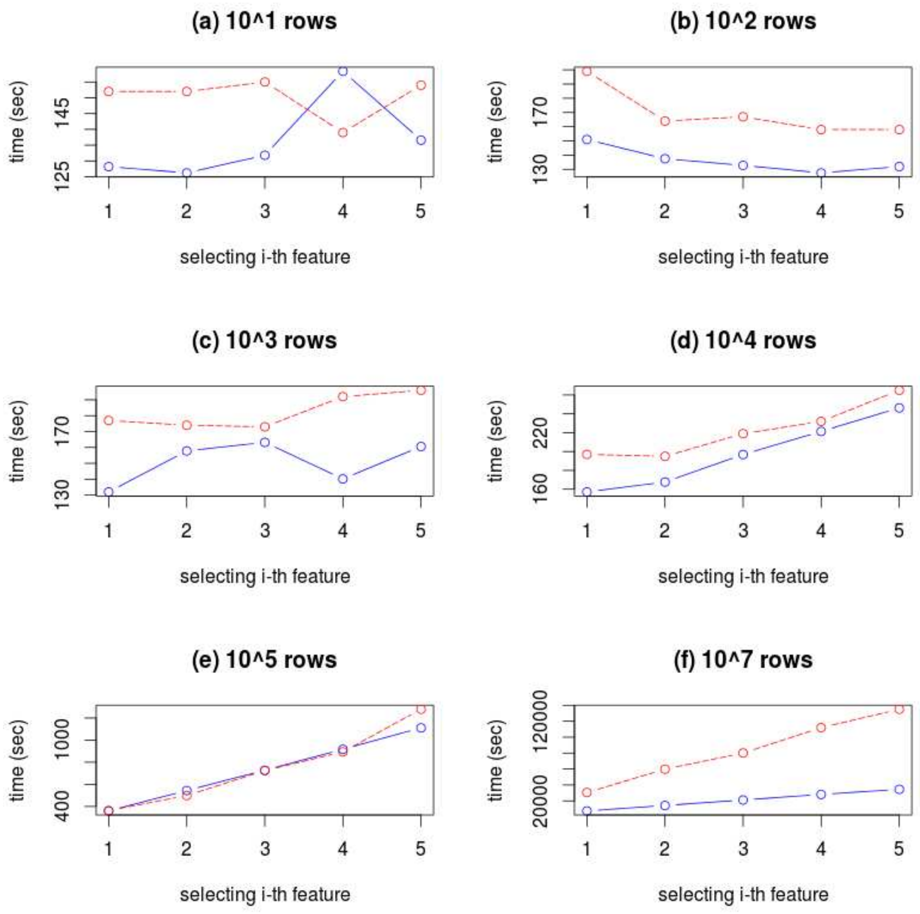


Figure E.1: mRMR: parallelism tests with 4 nodes. The red dashed line represents tests with one map task running, while the blue solid one the number of map tasks is decided by Apache Hadoop.

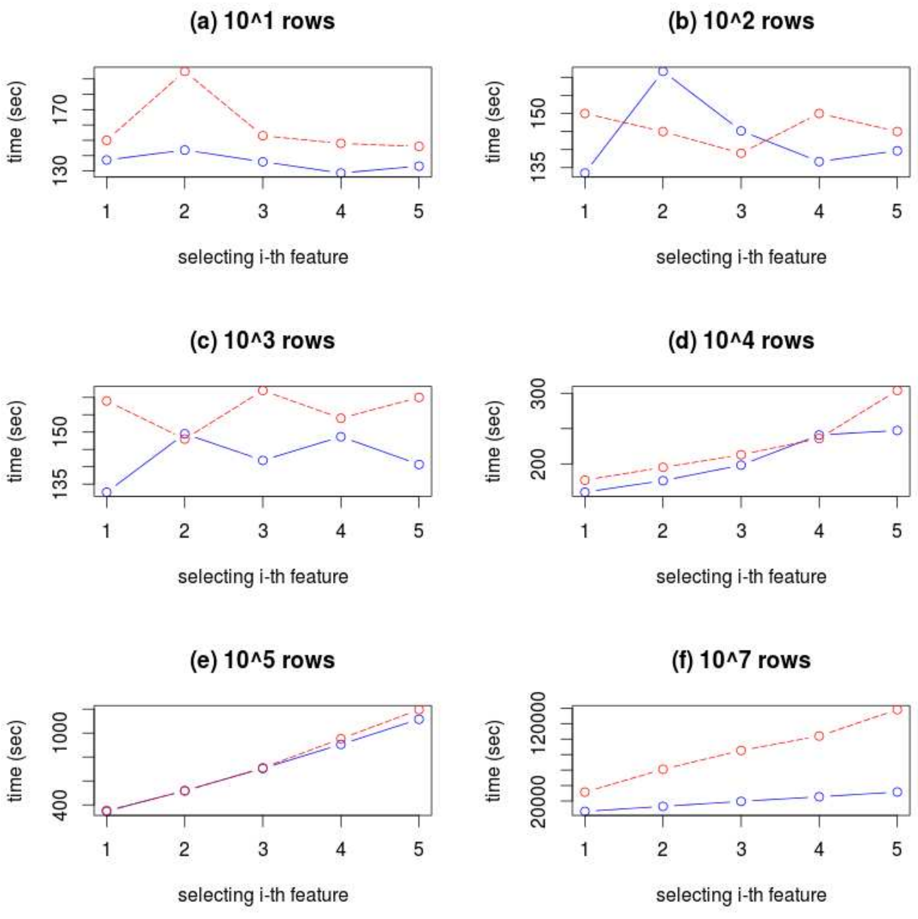


Figure E.2: mRMR: parallelism tests with 6 nodes. The red dashed line represents tests with one map task running, while the blue solid one the number of map tasks is decided by Apache Hadoop.

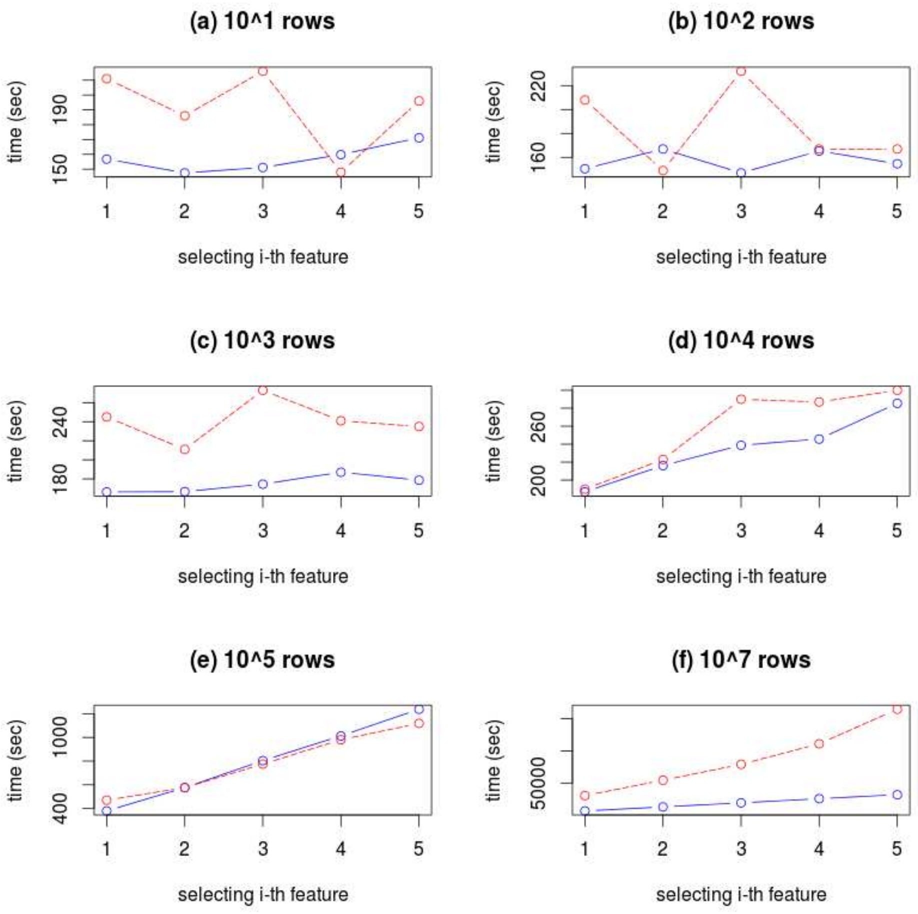


Figure E.3: mRMR: parallelism tests with 8 nodes. The red dashed line represents tests with one map task running, while the blue solid one the number of map tasks is decided by Apache Hadoop.

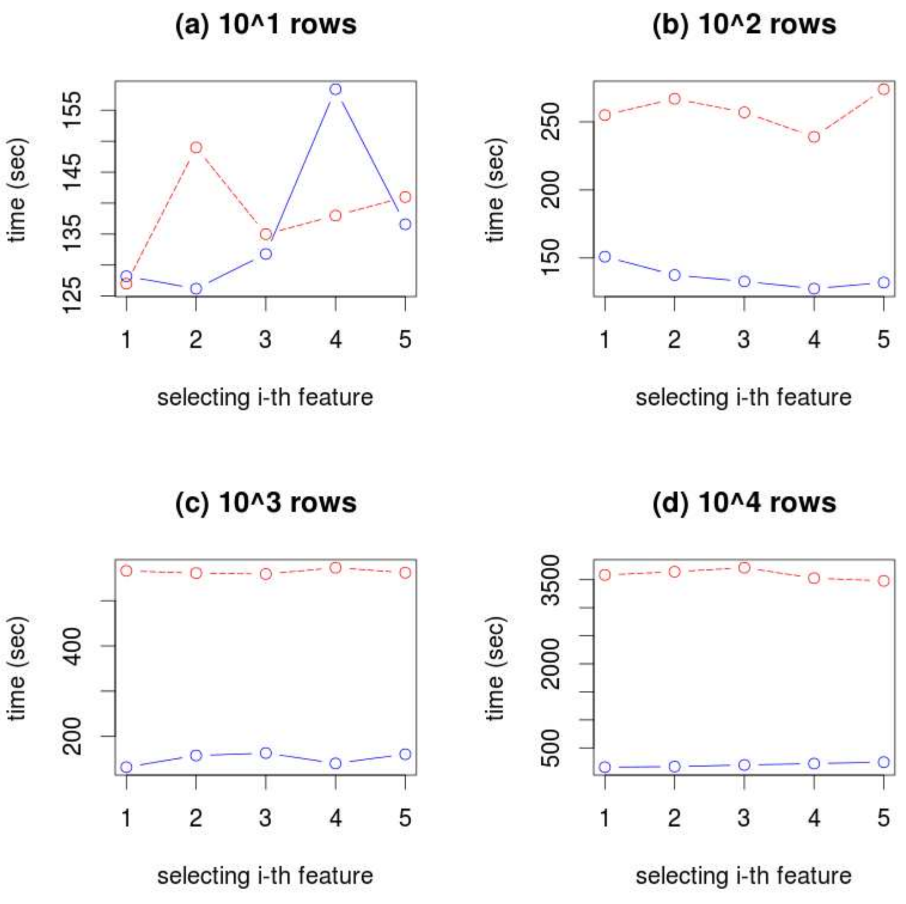


Figure E.4: mRMR: parallelism tests with 4 nodes. The red dashed dashed line represents tests with the maximum number of map task running, while the blue solid one the number of map tasks is decided by Apache Hadoop.

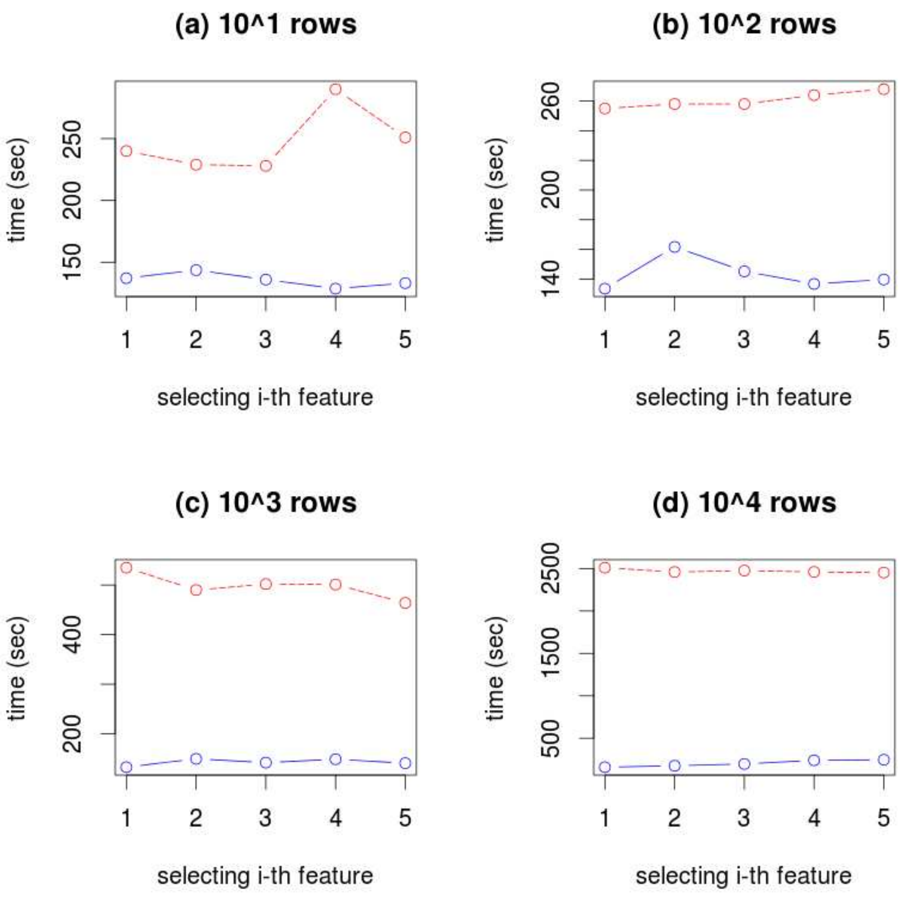


Figure E.5: mRMR: parallelism tests with 6 nodes. The red dashed line represents tests with the maximum number of map task running, while the blue solid one the number of map tasks is decided by Apache Hadoop.

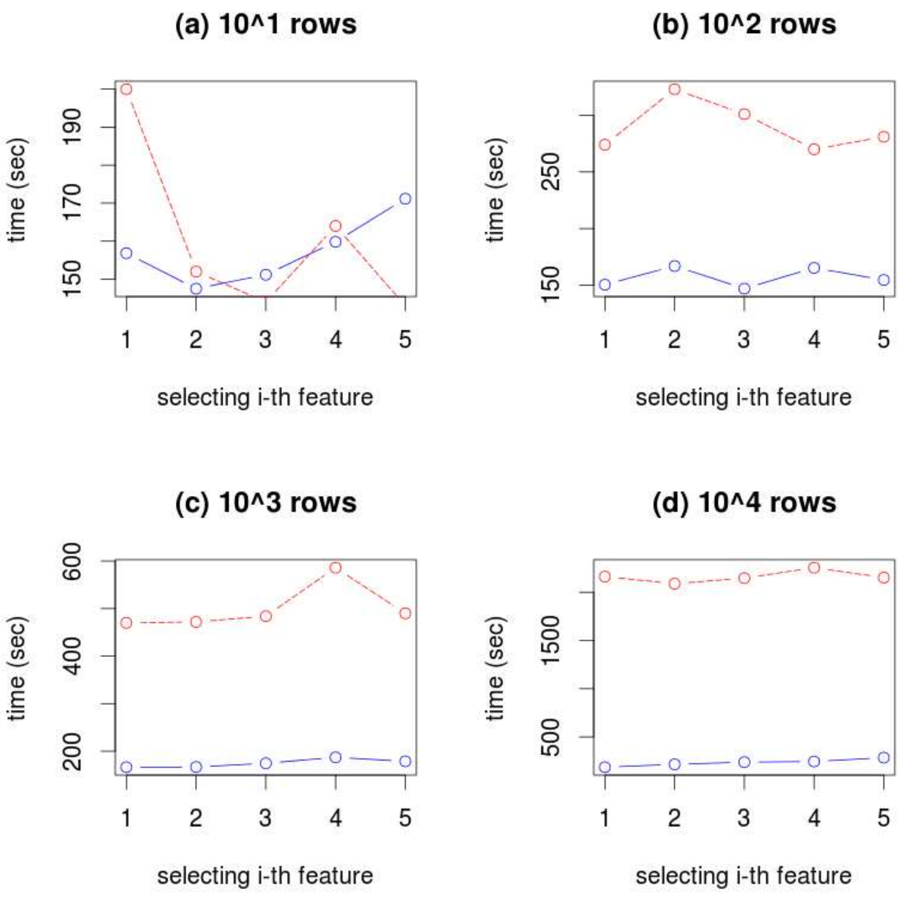


Figure E.6: mRMR: parallelism tests with 8 nodes. The red dashed line represents tests with the maximum number of map task running, while the blue solid one the number of map tasks is decided by Apache Hadoop.

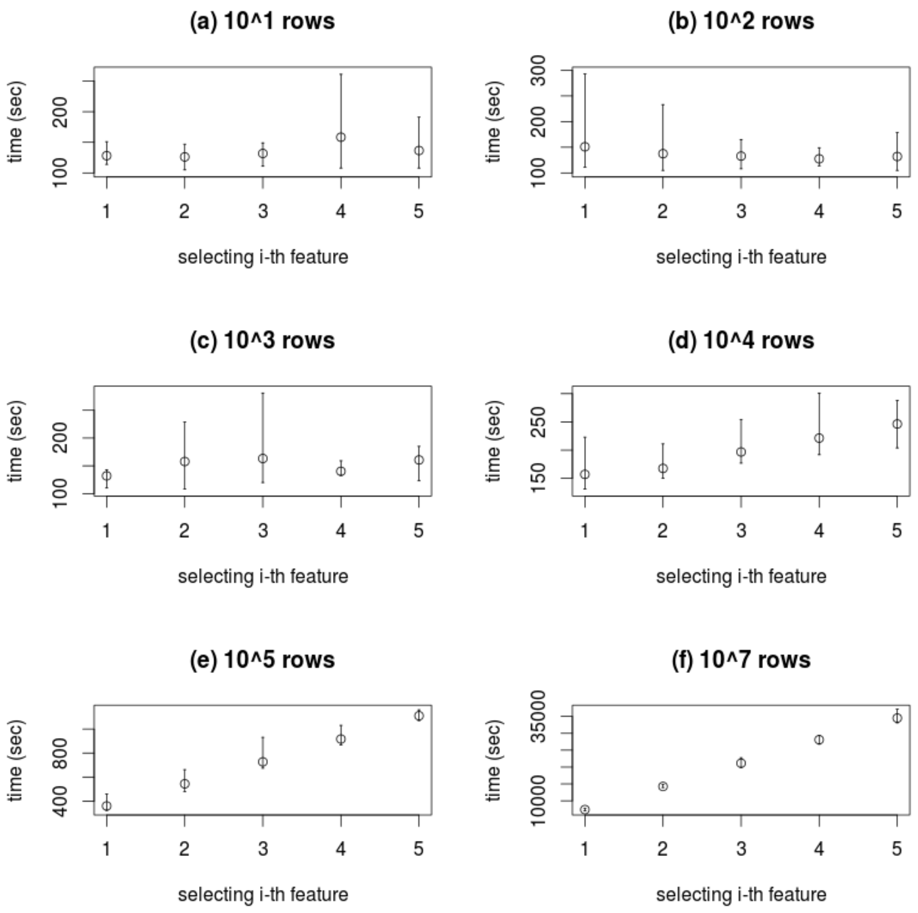


Figure E.7: mRMR: scalability tests with 4 nodes. The number of map tasks is set by Apache Hadoop.

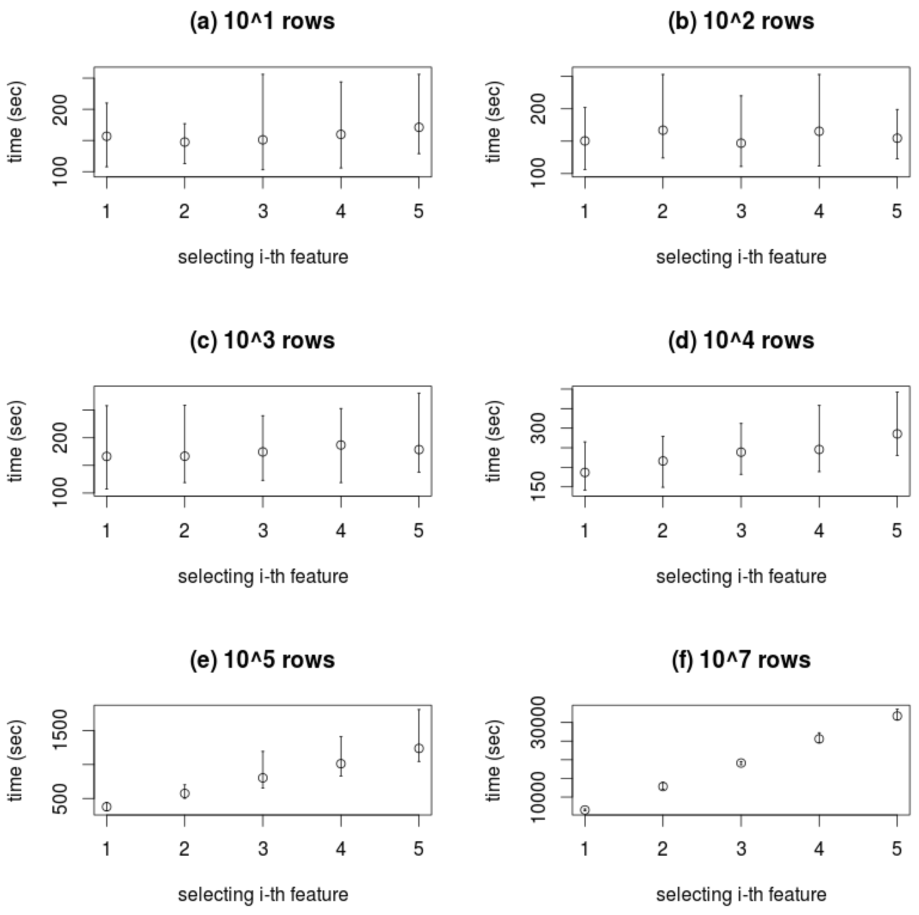


Figure E.8: mRMR: scalability tests with 8 nodes. The number of map tasks is set by Apache Hadoop.

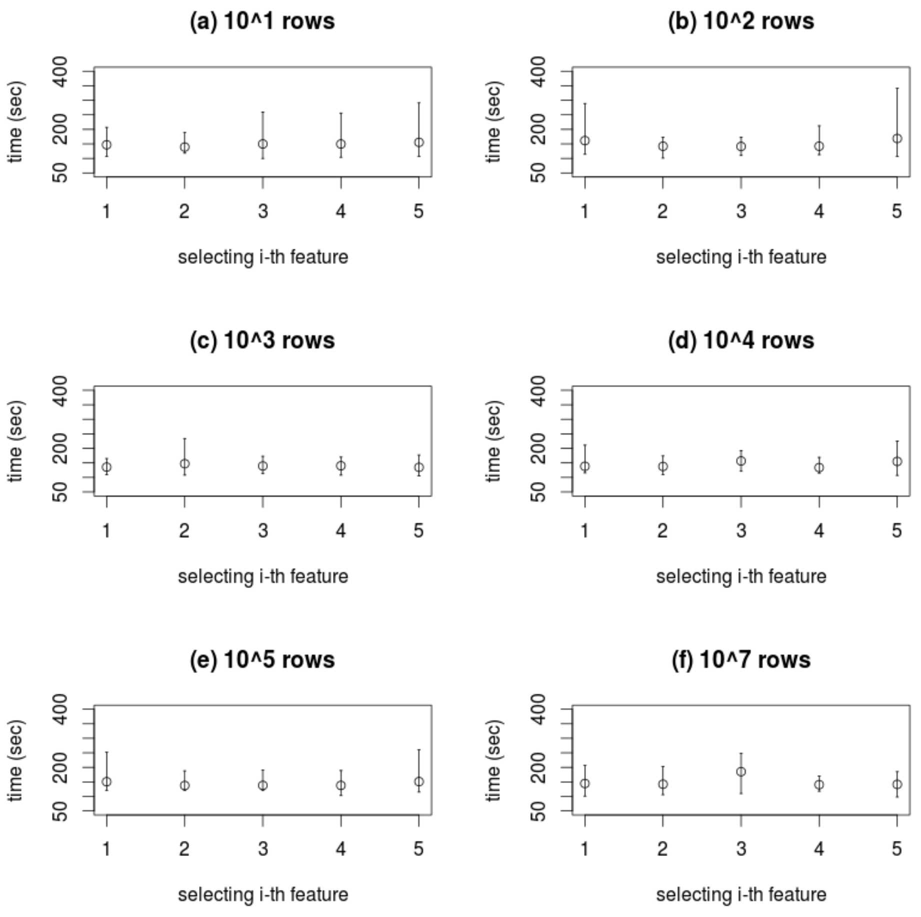


Figure E.9: Best feature: scalability tests with 4 nodes. The number of map tasks is set by Apache Hadoop.

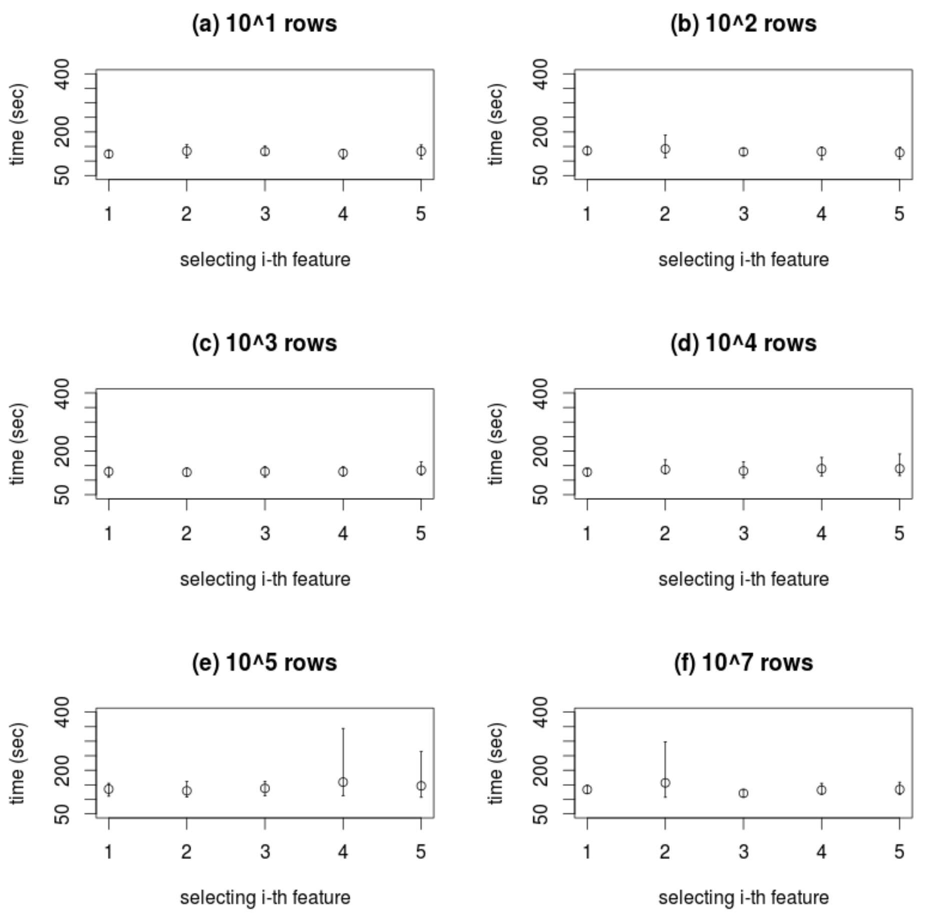


Figure E.10: Best feature: scalability tests with 6 nodes. The number of map tasks is set by Apache Hadoop.

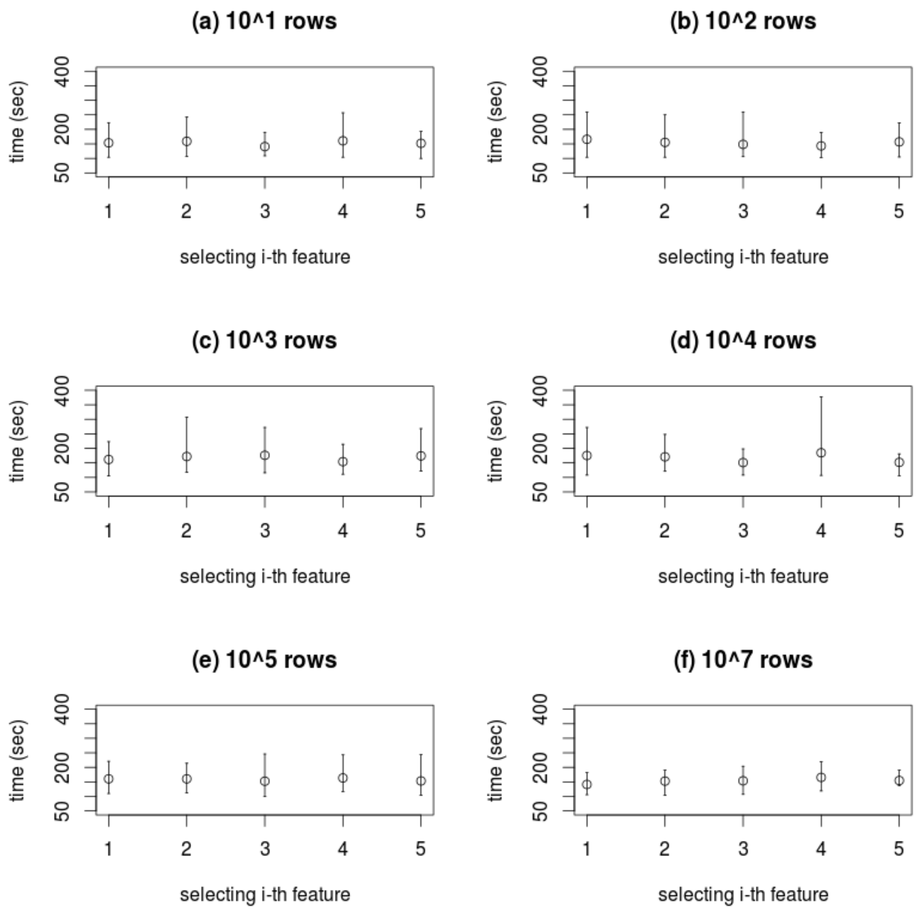


Figure E.11: Best feature: scalability tests with 8 nodes. The number of map tasks is set by Apache Hadoop.