Progress Report on “Big Data Mining”

Abstract

*Big Data* consists of voluminous, high-velocity and high-variety datasets that are increasingly difficult to process using traditional methods. *Data Mining* is the process of discovering knowledge by analysing raw datasets. Traditional Data Mining tools, such as Weka and R, have been designed for single-node sequential execution and fail to cope with modern Big Data volumes. In contrast, distributed computing frameworks such as Hadoop and Spark, can scale to thousands of nodes and process large datasets efficiently, but lack robust Data Mining libraries.

This project aims to combine the extensive libraries of Weka with the power of the distributed computing frameworks Hadoop and Spark. The system aims to achieve scalability to large volumes by partitioning big datasets and executing Weka algorithms against partitions in parallel.

Both frameworks support the MapReduce paradigm. In MapReduce, Map functions process dataset partitions in parallel and Reduce functions aggregate the results. Weka learning algorithms can be enclosed in classes (wrappers) that implement the Map interface and generate models on dataset partitions in parallel. Weka Meta-Learners can be enclosed in the Reduce interface and aggregate these models to a single output.

Weka wrappers for the first version of Hadoop, that already exist in Weka packages, were edited and compiled against the second (latest) version. A Hadoop2 cluster was built locally for testing and the system was tested in a variety of classification tasks. The system was then installed on AWS to carry out experiments at larger scales. Preliminary results demonstrate linear scalability.

The Spark framework was installed locally and was tested for interoperability with Hadoop MapReduce tasks. As expected since both systems are Java-based, Hadoop tasks can be executed on both systems and the existing solution is possible to be used in Spark.

The final part of the project will use this observation and implement wrappers for Weka algorithms on Spark. By taking advantage of its main-memory caching mechanisms, it is possible to greatly improve system performance.
## Contents

1. **Introduction** ..................................................................................................................4  
   1.1 Distributed Systems Processing .................................................................................4  
   1.2 Data Mining Tools .......................................................................................................5  
   1.3 Project Aim ..................................................................................................................5  
   1.4 Report Structure ..........................................................................................................5  

2. **Background** ..................................................................................................................6  
   2.1 Data Mining ..................................................................................................................6  
      2.1.1 Classification .......................................................................................................6  
      2.1.2 Clustering ..........................................................................................................7  
      2.1.3 Association Rules ..............................................................................................7  
      2.1.4 Weka ...................................................................................................................8  
      2.1.5 R .........................................................................................................................8  
      2.1.6 Data Mining and Parallel Performance ...............................................................8  
   2.2 Cluster Programming Models .......................................................................................8  
      2.2.1 MapReduce ........................................................................................................9  
      2.2.2 Hadoop ..............................................................................................................9  
      2.2.3 Beyond Hadoop ................................................................................................10  
      2.2.4 Iterative MapReduce ...........................................................................................11  
      2.2.5 Resilient Distributed Datasets ............................................................................11  
   2.3 Data Mining on Clusters ...............................................................................................13  
      2.3.1 Data Mining on MapReduce ...............................................................................13  
         2.3.1.1 Mahout .......................................................................................................13  
         2.3.1.2 Radoop .......................................................................................................13  
         2.3.1.3 R on MapReduce .......................................................................................13  
            2.3.1.3.1 Ricardo ............................................................................................14  
            2.3.1.3.2 SystemML .........................................................................................14  
         2.3.1.5 Weka on MapReduce ...................................................................................15  
      2.3.2 MLBase .................................................................................................................16  
   2.4 Summary ......................................................................................................................17  

3. **System Implementation** ..............................................................................................17  
   3.1 System Requirements .................................................................................................17  
   3.2 System Architecture ....................................................................................................18  
   3.3 Execution Model .........................................................................................................19  
   3.4 Spark and Main-memory Caching ..............................................................................20  
   3.5 Evaluation ...................................................................................................................21  

4. **Progress** ......................................................................................................................22  
   4.1 Setting Up the Hadoop Environment ............................................................................22  
      4.1.1 Local Testbed .....................................................................................................23  
      4.1.2 AWS ..................................................................................................................23
4.2 Weka on Hadoop .................................................................................................................... 23
4.3 Preliminary Test Results ........................................................................................................ 23
  4.3.1 On Local Testbed ........................................................................................................ 24
    4.3.1.1 Building Headers ............................................................................................ 25
    4.3.1.2 Classifier Training ....................................................................................... 26
    4.3.1.3 Classifier Evaluation .................................................................................... 27
  4.3.2 AWS .............................................................................................................................. 27
4.4 Spark ....................................................................................................................................... 27
4.5 Plan ......................................................................................................................................... 28

5. Conclusion ................................................................................................................................... 29

List of Figures

1.1 Cluster Architectures ............................................................................................................. 4
2.1 Data Mining Process ............................................................................................................ 6
2.2 Supervised Learning Process ............................................................................................. 7
2.3 MapReduce Execution Overview .................................................................................... 9
2.4 Hadoop Stack ..................................................................................................................... 10
2.5 HaLoop and MapReduce .................................................................................................. 11
2.6 RDD Lineage Graph .......................................................................................................... 12
2.7 Transactions and Actions on RDDs ................................................................................. 13
2.8 Ricardo .................................................................................................................................. 14
2.9 Distributed Weka ............................................................................................................... 15
2.10 MLBase Architecture ....................................................................................................... 16
3.1 System architecture ............................................................................................................. 19
3.2 Weka on Hadoop execution model .................................................................................. 20
4.1 Building Headers ............................................................................................................... 24
4.2 Linear Regression Training ............................................................................................... 25
4.3 Classifier Evaluation ......................................................................................................... 25
4.4 Classifier Training on AWS ............................................................................................. 26

List of Tables

4.1: Implementation Plan .......................................................................................................... 28
1. Introduction

Datasets across all domains are increasing in size exponentially [1]. Short et al. [2] estimated that in 2008 servers worldwide processed 10x10^{21} bytes (zetta-bytes) and expected this number to double every two years. These developments created the term “Big Data”.

According to Gartner, Big Data is “high-volume, high-velocity, and/or high-variety information assets that require new forms of processing to enable enhanced decision making, insight discovery and process optimization” [3]. In practice, they are datasets that are increasingly difficult to collect and process using traditional methods.

Another aspect that drives the development of big data technologies is the emerging trend of data-driven decision-making. For example, McKinsey [4] calculated that the health care system in USA could save up to $300bn by better understanding of related data (clinical trials, health insurance transactions, wearable sensors etc.). This trend requires processing techniques to transform data to valuable insights. The field that deals with knowledge extraction from raw data is known as Data Mining.

Developments mentioned above are closely tied-in with the evolution of distributed systems. Due to large volumes, processing is performed by organised clusters of computers. Proposed cluster architectures include shared-memory, shared-disk and shared-nothing. Shared-memory clusters have a global main-memory shared between processors by a fast interconnect. Shared-disk clusters use an array of disks accessible through the network and each processor has its own private memory. In shared-nothing clusters every node has a private set of resources. The following figure by Fernandez [5] presents these architectures.

![Figure 1.1: Cluster Architectures [5]](image)

All but the latter suffer difficulties in scaling [6]. Pioneered by Google, shared-nothing architectures have dominated mainly because they can scale dynamically by adding more inexpensive nodes. Emerging cloud computing providers, such as AWS (Amazon Web Services) [7], offer access to these architectures on demand.

1.1 Distributed Systems Processing

Google in 2004 [8] introduced MapReduce, a programming model targeting shared-nothing architectures. MapReduce expresses computations using two operators (Map and Reduce), schedules their execution in parallel on dataset partitions and guarantees fault-tolerance through replication. The Map operator transforms dataset partitions in parallel and the Reduce operator aggregates the results.
Yahoo in 2005 released an open source implementation of MapReduce called Hadoop [9]. The following years, many institutions adopted Hadoop [10] and many others plan Hadoop integration in the near future [11].

Although the MapReduce paradigm can express many data mining algorithms efficiently [12], improvements are possible: a loop operator could better express iterative algorithms and main-memory caching would enable faster data access. What is more, data mining projects that use Hadoop could benefit from scalable data mining libraries.

Out of many possible options [13], this project uses Spark [14] as a target platform. Spark supports main-memory caching (faster data access), it has a loop-aware scheduler (better support for iterative algorithms) and it is Hadoop compatible. Additionally, Spark is implementing the MapReduce paradigm and it is Java-based. These features enable users to re-use Hadoop applications in Spark. Spark outperforms Hadoop up to two orders of magnitude in many cases [14].

1.2 Data Mining Tools

Weka and R are two of the most popular data mining tools produced by the open-source community [17].

Weka contains libraries that cover all major categories of data mining algorithms and has been under development for more than 20 years [15]. However, they were developed targeting sequential single-node execution and not for distributed environments. Consequently, Weka can only handle small datasets.

R [16] is a programming language designed for statistical computing. It incorporates essential data mining components such as linear and non-linear models, statistical testing and classification among others. It also provides a graphical environment for results visualisation. Although it is possibly the most popular tool [17] in the field, it demonstrates the same shortcomings as Weka.

Weka was selected as the project data mining framework due to the fact that is written in Java (R is C-based) and it is native to the Java-based execution environment of Hadoop and Spark.

1.3 Project Aim

This project aims to integrate Weka's data mining libraries with the power of the distributed computing frameworks Hadoop and Spark. This will be achieved by wrapping Map and Reduce interfaces around Weka's algorithms. More specifically, Map tasks will execute Weka's algorithms against dataset partitions in parallel and generate partial models. Reduce tasks will use Weka's model aggregation classes (Meta-Learners) to produce the final output.

Solution evaluation will test the system against datasets of varying sizes and computer clusters of varying number of nodes. The aim is to measure weak and strong scaling [18] of the system in varying problem parameters (algorithms, dataset size, number of nodes). More specifically, linear strong scaling (if solution time decreases as the nodes increase at a fixed problem) and linear weak scaling (if solution time is constant at a fixed per node problem) would indicate that the system demonstrates scalability as data volumes increase.

Other important factors are resource utilisation (mainly network resources which can be a bottleneck [19]), and start-up/clean-up overhead (time consumed to initialise and terminate the system).

1.4 Report Structure

Chapter 2 presents an overview of data mining algorithms and distributed computing
frameworks. Chapter 3 describes the system architecture, the execution model and the evaluation methodologies. Chapter 4 presents the implementation progress and preliminary results. Finally, Chapter 5 draws the conclusions so far.

---

2. Background Overview

This chapter overviews data mining techniques (2.1), presents modern cluster programming models (2.2) and distributed data mining efforts using these models (2.3).

2.1 Data Mining

Data mining procedures include a number of basic steps: data recording, noise reduction, data analysis and data representation and interpretation. Figure 2.1 illustrates these steps.

![Diagram of the Data Mining Process](image)

*Figure 2.1: The Data Mining Process*

High data volumes lead to signal to noise ratios that can reach 1:9 [20]. Consequently, noise reduction phase is substantial to maximize data quality and to minimize noise effects. Data analysis phase can be divided into four major categories of algorithms: Classification, Clustering, Association Rules and Mining Emerging Data-types (Text, Graphs and Streams).

As mentioned above, the implementation will be based on Weka core libraries that mainly support the first three. Consequently, this project will focus on Classification, Clustering and Association rules.

The following sections describe the main characteristics of these algorithms in brief.

2.1.1 Classification

Classification is a process in which a hypothesis function (classifier) infers the category in which a data object belongs. In supervised learning, a learning algorithm processes a dataset with labeled data (data objects whose category (class) is known) and extracts the function's parameters. The classifier can then be used to predict the class of unknown data objects.

The following figure taken from Andrew Ng's online class [21] illustrates the process described above.
Classifiers generated from the same dataset can be combined to form a voted ensemble using Melta-Learners [22]. In these ensembles the classification result is based on the prediction of the majority.

2.1.2 Clustering

Clustering is the process in which data objects are grouped together in classes based on some measure of similarity. It is different from classification because data classes are not known. It is also known as unsupervised learning and is used to discover hidden structure in datasets.

Clusterers can be divided in iterative and incremental. Iterative algorithms require multiple passes over the whole dataset to converge. At incremental cluster construction [23] two approaches can be applied: a) adding data points at each iteration and recomputing the cluster centers and b) adding a cluster center at each iteration. In both cases, the solution is built incrementally and it is possible to find a near optimal solution in a single pass.

Consensus clustering [24] is the process of combining different clusterings of the same dataset in a single output.

2.1.3 Association Rules

Association rule learning discovers correlations and associations between items in an itemset. Given the large amounts of data stored by retailers, association rules emerged as a solution to the market-basket analysis problem [25].

A bit vector is used to indicate the presence or absence of an item in an itemset. A group of bit vectors is used to represent a set of transactions. By analysing the vectors, it is possible to discover items that frequently occur together. These frequent occurrences are expressed in the form of rules. For example the rule:

\{ cheese, bread \} => \{ burger\}

Indicates that if a customer buys cheese and bread, will probably also buy burgers. However, not all rules are interesting. Some measure of significance is needed. Support (percentage of transactions that contain cheese, bread and burger), Confidence (percentage of transactions containing cheese and bread that also contain burgers) and Lift (a correlation measure of the two itemsets) are three popular measures. By setting thresholds to these measures it is possible to discover interesting rules on a set of transactions using various algorithms.
2.1.4 Weka

Weka [15] includes well tested and extensively reviewed implementations of most popular data mining algorithms mentioned above. It contains tools that support all phases of the data mining procedure.

A large collection of filters can be used to pre-process datasets and reduce noise. Algorithms spanning across all major categories can be used for data analysis. It offers a GUI (Graphical User Interface) that supports interactive mining and results visualization. Finally, it automatically produces statistics to assist results evaluation.

The major issue with Weka is that it only supports sequential single-node execution. As a result, the size of the datasets that Weka can handle in the existing implementation is limited.

2.1.5 R

R [16] is a C-based statistical programming language with built-in support for linear and non-linear modelling, matrix manipulation, time-series analysis, data cleaning, statistical testing and graphics among others [26]. It is interpreted and can be used interactively to implement data mining tasks. Statistical procedures are exposed through simple commands. It gained popularity [17] among analysts mainly because it does not demand advanced programming expertise. However, it is designed for sequential execution and suffers the same shortcomings as Weka in Big Data problems.

2.1.6 Data Mining and Parallel Performance

Data Mining algorithms can be either single-pass or iterative [27]. Single-pass algorithms have an upper bound in execution times. Iterative algorithms loop over the dataset until a stop condition is met (convergence) and thus execution times may vary. Due to the sizes of the datasets, this project deals only with partitioned parallelism: the dataset is partitioned and computing nodes process the partitions in parallel.

For single pass algorithms, this method can theoretically yield speed-up analogous to the number of nodes. In practice, an overhead associated with distributing computations and aggregating the results over all partitions will set the limit a little lower. However, this overhead can be experimentally computed and system performance is predictable. An example of this case could be computing class means and variances and building a Gaussian model for Naive Bayes. Each node can compute the statistics for each class at its local chunk in one pass and aggregation requires a single synchronisation step.

In contrast, iterative algorithms can be unpredictable. The number of iterations that the system will need to converge cannot be defined from the start. Two different approaches are possible [57]: synchronous and asynchronous.

In synchronous, each node computes the model's parameters on its own chunk in a single iteration. Then a synchronisation step is used to gather local parameters from all nodes and aggregate them to global values. During synchronisation, the stop condition is also evaluated. If not reached, each node obtains a copy of the global values and begins a new iteration. This technique achieves load balancing between the nodes, but requires constant node communication and as mentioned above [19], network can be a bottleneck.

In asynchronous, each node computes a local model and a single synchronisation step at the end aggregates the results. This technique minimises network overheads but load balancing is not guaranteed. Nodes that struggle to converge will slow down the performance of the system. One
solution is to enforce a deadline: each node has a certain number of iterations to meet the stop condition. After the deadline, the final model will be computed only on the nodes that managed to converge. This technique may lack precision, but the execution time is guaranteed and speed-up is analogous to the nodes.

This project will study asynchronous techniques only, for two main reasons:
1. Weka does not support synchronisation by default and adding this feature would require re-implementing all the algorithms from scratch.
2. Modern distributed computing frameworks described in the next section suffer from network bottlenecks [19] and developers seek to minimise network use.

2.2 Cluster Programming Models

Cluster programming models provide an interface between distributed computing frameworks and algorithms. The following sections present a number of important efforts.

2.2.1 MapReduce

MapReduce was introduced by Google [8] in order to tackle the problem of large-scale processing in clusters of inexpensive nodes. Datasets in MapReduce are automatically partitioned, replicated and distributed across the cluster. This practice ensures that partitions can be processed in parallel and fault-tolerance can be guaranteed through replication.

A MapReduce cluster consists of a Master node which handles data partitioning and schedules tasks automatically in an arbitrary number of Workers. The Master also holds meta-data concerning partition locations in the cluster. This practice forces Workers to process their local partition and avoids transmitting large data chunks through the network.

The user specifies two functions: Map and Reduce. Map is used to filter and transform a list of key-value pairs into intermediate key-value pairs. Reduce processes the intermediate pairs, aggregates the results and produces the output.

Once the user has specified the Map and Reduce functions, the runtime environment automatically schedules the execution of Mappers on idle cluster nodes. Each node executes the Map function against its local dataset partition, writes intermediate results to its local disk and periodically notifies the Master of its progress. As the Mappers start producing intermediate results, the Master node starts assigning Reduce tasks to (other) idle cluster nodes. Each intermediate result has a key and it is distributed to the Reducer that handles this key (or key range).

Figure 1 in [8] presents the procedure above.
2.2.2 Hadoop

Hadoop [9] is an open source implementation of MapReduce. It was built by Yahoo and released under the Apache License in 2006.

Its main components are HDFS (Hadoop Distributed File System) [28], YARN (Yet Another Resource Negotiator) [29] and the MapReduce framework. HDFS is disk-based file system that spans across all cluster nodes. YARN is responsible for managing cluster resources and provide applications with execution containers for Map and Reduce tasks. MapReduce is a set of libraries that implement the MapReduce paradigm described above.

Hortonworks' webpage [30] presents the system described above in the following figure.

Figure 2.3: MapReduce execution overview [8]
2.2.3 Beyond Hadoop

As briefly discussed earlier, Hadoop has room for improvement:

- A loop operator: Many data mining algorithms require multiple iterations over a dataset to converge to a solution. For example, the K-Means clustering algorithm iterates over a dataset until cluster assignments remain unchanged after two successive iterations. These iterations are usually included in the driver program that the user provides. The main issue is that the dataset needs to be reloaded at every step from secondary storage (with large I/O cost). If Hadoop was loop-aware, it could avoid reloading data and restarting job threads [31].

- HDFS is a disk based filesystem. However, modern clusters possess main memory that can exceed 1TB and most mining tasks are within this limit [32]. As a result, significant performance improvement is possible by caching datasets to main memory instead of reloading portions of it from disk.

- Scalable data mining libraries: Mahout [33] was designed to operate on top of Hadoop. It has a complicated programming interface and demonstrated poor scalability on many tasks [34]. These issues lead to numerous efforts towards novel systems. The following sections present a number of important projects in the area.

2.2.4 Iterative MapReduce

HaLoop [31] introduced a loop operator to the MapReduce framework aiming to provide built-in support for iterative algorithms. HaLoop's scheduler is loop-aware and its caching mechanisms cache loop-ivariant data.
Figure 2 in [31] shows the boundaries between user application and system code in MapReduce and HaLoop.

Stop conditions are evaluated in the system code and do not need an extra job as in MapReduce. The loop-aware scheduler co-locates jobs that use the same data in successive iterations. The caching mechanism is used to save loop-invariant data between iterations. These developments reportedly [31] provide up to 85% speed-up compared to Hadoop on iterative algorithms. However, non-iterative algorithms demonstrate similar performance with Hadoop and the programming interface is significantly more complicated.

2.2.5 Resilient Distributed Datasets - Spark

Resilient Distributed Datasets (RDDs) [14] are a distributed main-memory abstraction that enables users to perform in-memory computations in large clusters. They are implemented in the open-source Spark [35] framework.

RDDs are an immutable collection of records distributed across the main memory of the cluster. These data structures can be created by invoking a set of operators either on non-volatile memory data or on other RDDs. The system logs dataset transformations using a lineage graph. The system is “lazy”: it does not materialise transformations until the user requests either an output or saving changes to persistent storage.

Figure 1 in [14] illustrates an RDD lineage graph.

Figure 2.5: HaLoop and MapReduce [31]

Figure 2.6: RDD lineage graph: Boxes represent RDDs and arrows transformations. [14]
Operators are divided into two categories: Transformations and Actions. Transformations define a new RDD based on existing RDDs. Actions materialise the Transformations and either return a value to the user or export data to secondary storage.

Since it was inspired by MapReduce, Transformations include Map and Reduce operators. The system was designed to tackle the shortcomings of MapReduce in iterative computations and as a result it is loop-aware. Other operators include join, union, crossProduct and groupBy. These enable the use of the system as an Online Analytical Processing (OLAP) engine as well.

Table 2 in [14] provides the complete set of operators and actions on RDDs currently supported by the Spark framework.

<table>
<thead>
<tr>
<th>Transformations</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>map(f : T ⇒ U)</td>
<td>RDD[U] ⇒ RDD[U]</td>
</tr>
<tr>
<td>filter(f : T ⇒ Bool)</td>
<td>RDD[T] ⇒ RDD[T]</td>
</tr>
<tr>
<td>flatMap(f : T ⇒ Seq[U])</td>
<td>RDD[T] ⇒ RDD[U]</td>
</tr>
<tr>
<td>sample(fraction : Float)</td>
<td>RDD[T] ⇒ RDD[T]</td>
</tr>
<tr>
<td>groupByKey()</td>
<td>RDD[(K, V)] ⇒ RDD[K, Seq[V]]</td>
</tr>
<tr>
<td>reduceByKey(f : (V, V) ⇒ V)</td>
<td>RDD[(K, V)] ⇒ RDD[K, V]</td>
</tr>
<tr>
<td>union()</td>
<td>(RDD[T], RDD[T]) ⇒ RDD[T]</td>
</tr>
<tr>
<td>join()</td>
<td>(RDD[(K1, V1)], RDD[(K2, W2)]) ⇒ RDD[(K1, (V1, W2))]</td>
</tr>
<tr>
<td>crossProduct()</td>
<td>(RDD[T], RDD[U]) ⇒ RDD[T, U]</td>
</tr>
<tr>
<td>mapValues(f : V ⇒ W)</td>
<td>RDD[(K, V)] ⇒ RDD[K, W]</td>
</tr>
<tr>
<td>sort(e : Comparator[K])</td>
<td>RDD[(K, V)] ⇒ RDD[(K, V)]</td>
</tr>
<tr>
<td>partitionBy(p : Partitioner[K])</td>
<td>RDD[(K, V)] ⇒ RDD[K, V]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Actions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>count()</td>
<td>RDD[T] ⇒ Long</td>
</tr>
<tr>
<td>collect()</td>
<td>RDD[T] ⇒ Seq[T]</td>
</tr>
<tr>
<td>reduce(f : (T, T) ⇒ T)</td>
<td>RDD[T] ⇒ T</td>
</tr>
<tr>
<td>lookup(k : K)</td>
<td>RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)</td>
</tr>
<tr>
<td>save(path : String)</td>
<td>Outputs RDD to a storage system, e.g., HDFS</td>
</tr>
</tbody>
</table>

Figure 2.7: Transformations and Actions available on RDDs in Spark.[14]

In order to execute tasks in Spark, the user needs to provide a driver program which defines a path to secondary storage and a set of Transformations and Actions. The system proceeds to create an RDD and distribute its records across the main-memory of the cluster nodes. When an Action is issued the system will proceed to schedule the execution of the requested Transformations. Each Node at the cluster will process its local set of records and return the results. By caching datasets to main memory the system avoids slow disks reads and can perform up to 100 times faster than Hadoop's MapReduce [14] on iterative algorithms. In cases where the dataset is larger than the available amount of main memory, Spark has a mechanism to flush portions of the dataset to secondary storage. In general, memory consumption is a configuration parameter in Spark and the mechanism can be triggered at a user-defined level. Even when minimal memory is available, Spark outperforms Hadoop due to shorter initialisation and termination overheads [36].

Spark is Hadoop compatible: it uses HDFS as a non-volatile data source and YARN as a resource negotiator between user programs. It can be installed to a Hadoop cluster and share resources with existing Hadoop applications. Spark's Scala [37] API supports closures and can pass Hadoop Map and Reduce tasks as arguments to its native operators. This technique can enable users to deploy existing Hadoop MapReduce tasks to Spark with minor adjustments.
2.3 Data Mining on Clusters

This section outlines efforts made to express algorithms presented in (2.1) using the models presented in (2.2).

2.3.1 Data Mining with MapReduce

Chu [12] demonstrated that many data mining algorithms can be expressed using the MapReduce paradigm efficiently. The following years, many efforts were made towards building data mining systems on top of Hadoop. The following sections present the most notable.

2.3.1.1 Mahout

Mahout was a side project of Hadoop aiming to provide scalable Data Mining libraries. It was community-based and users submitted their MapReduce implementations of various algorithms. Since the libraries did not provide a general framework for building algorithms, the quality of provided solutions varied. This lead to poor performance [34], inconsistencies in the content of its various releases [33] and it was eventually discontinued.

2.3.1.2 Radoop

Radoop [38] introduced the RapidMiner [39] toolkit to Hadoop. RapidMiner has a graphical interface to design work-flows which consist of data-loading, cleaning, data-mining and visualization tasks. Radoop introduced operators that can read data from HDFS and then execute mining tasks against them. However, the new operators neither use RapidMiner mining libraries nor its in-memory capabilities. Radoop operators correspond to Mahout algorithms. Consequently, at runtime the workflow designed by the user gets translated to Mahout tasks and gets executed on a Hadoop cluster.

The system has an easy-to-use interface and demonstrates basic scalability. However, it is essentially a user interface for Mahout and suffers from the shortcomings mentioned above.

2.3.1.3 R on Hadoop

The next sections present two notable efforts aiming to combine the widely used statistical language R and the power of Hadoop.

2.3.1.3.1 Ricardo

Ricardo [40] is an early effort to introduce R to distributed computing frameworks. The system used the declarative scripting language Jaql [41] and Hadoop to execute R programs in parallel. It tried to address the problem that MapReduce was low-level and thus difficult for most analysts.

The system consists of three components: the user interface where the analyst writes mining tasks using R and a small set of Jaql functions; an R-Jaql bridge that integrates R programs to Jaql declarative queries; and the Jaql compiler that compiles the queries to a series of MapReduce jobs in the Hadoop cluster. Flow control is performed by the user program. The bridge allows R programs to make calls to Jaql scripts and Jaql scripts to run R processes on the cluster. The bridging was implemented using JNI (Java Native Interface).

Figure 3 in [40] presents the architecture.
The system uses R-syntax with which most analysts are familiar and has a simple interface. However, execution times are doubled compared to native MapReduce jobs for the same task. This is due to the overhead produced by compiling high-level declarative Jaql scripts to low-level MapReduce jobs.

2.3.1.3.2 SystemML

SystemML [42] is another system that tries to expose Hadoop through R syntax. It differs from Ricardo because it uses a heuristic to optimize MapReduce tasks before submitting them to the cluster. More specifically, the user writes programs using DML (Declarative Machine Learning); a declarative language that mimics the interface of R and exposes a set of linear algebra primitives and flow control operators (while, for). SystemML proceeds to break declarative DML scripts to smaller components named statement blocks. Each statement block is then analysed using algebraic rewrites and cost-based optimizations to produce a high-level execution plan. The optimized high-level plan is then compiled to a set of low-level physical plans that consist of MapReduce jobs. Using a greedy heuristic, the system groups together many low-level operations that use the same data and avoids redundant disk scans. At runtime a generic MapReduce job is used to execute multiple low-level plans. Depending on data characteristics, SystemML orchestrates execution dynamically by seeking more optimizations at runtime.

Results in [42] show that the system has linear scalability, outperforms hand-coded solutions and exposes a very simple programming interface. However, the system was designed by IBM and it is proprietary.

2.3.2 Distributed Weka

Early efforts made to introduce Weka to distributed environments include WekaG [43],
parallelWeka [44] and Weka4WS [45]. WekaG and WekaWS use web services to submit tasks to remote servers for execution. However, they do not support parallelism; each server executes an independent task on its own local data. ParallelWeka proposed parallel cross-validation where each server receives a dataset copy, computes a fold and sends back the results. This practice cannot be applied to large scales because of network bottlenecks.

Wegener et al. [46] introduced Weka to Hadoop. The goal was to merge the user-friendliness of Weka's user interface with Hadoop's ability to handle large datasets.

The system architecture consists of three actors: the Data Mining Client, the Data Mining Server and the Hadoop cluster. The client uses Weka's user interface to build mining tasks and then submits them to the server. The server receives the client's request, computes the sequential part of the algorithm locally and submits the parts that can be executed in parallel to the Hadoop cluster. These procedures require to review Weka libraries, identify the parts of each algorithm that can be parallelised and rewrite them using the MapReduce paradigm. In the server, Weka's data-loader was extended to avoid loading datasets to main-memory and instead perform a series of disk reads.

Figure 1 in [46] presents the architecture.

![Figure 2.9: Distributed Weka [46]](image)

This methodology would require to re-implement Weka's algorithms from scratch. Additionally, since clusters possess increasing amounts of memory, it would benefit performance-wise from main-memory caching (that Weka already supports).

2.3.3 MLBase

MLBase [47] is a novel distributed computing framework which aims to introduce query optimization techniques to data mining and machine learning algorithms.

The user can build data mining tasks using a high-level declarative language and submit them to the cluster's Master. The system then proceeds to parse the request to form a Logical Learning Plan. This plan consists of feature extraction, dimensionality reduction, filtering, learning and evaluation algorithms. The optimizer processes that plan using statistical models and heuristics. An Optimized
Learning Plan (OLP) is produced based on which combination of algorithms is likely to have better performance (execution time and accuracy). MLBase then proceeds to translate OLP to a set of primitive operators that the run-time environment supports. These include relational operators (joins, projects), filters and high-level functions like Map in MapReduce. The primitives are then scheduled for parallel execution in the cluster's workers.

The system proceeds to build the model in stages. From an early stage, it returns a preliminary model to the user and it continuous to refine it in the background.

Figure 1 in [47] illustrates the procedure.

The users of this system will be able to submit tasks without specifying an algorithm. The system will then parse the request and select a near-optimal solution by analysing various alternatives. This would be an important development since users would no longer need to find reliable, scalable and accurate solutions solely based on intuition.

As of May 2014 the system is still under development. However, the interfaces of its components were described in [34] and a proof-of-concept implementation using Spark was tested with impressive results.

**2.4 Summary**

Sequential solutions fail to cope with big data workloads. Hadoop is a field-tested solution for large datasets and it sets the standard for industrial big data platforms. However, Hadoop's native implementation of MapReduce is inefficient in implementing and executing iterative algorithms. Spark tackles this issue by introducing a main-memory caching mechanism and a loop-aware scheduler. Additionally, Spark is Hadoop compatible and can be easily installed in existing Hadoop clusters.
Sicne both Hadoop and Spark are Java-based, Weka can natively run on top of both systems without additional bridging overheads. Hadoop's HDFS will provide persistent storage and YARN will be used as a cluster resource manager. Spark's batch processing operators will be used as an execution environment for Weka classes. By executing Spark Map and Reduce tasks containing Weka classes against dataset partitions, it is possible to achieve very significant speed-up as opposed to traditional sequential solutions.

The following Chapter provides greater detail on how the system will be implemented.

3. System Development

The following sections describe the requirements (3.1), the architecture (3.2), the execution overview (3.3), the Spark integration (3.4) and the evaluation plan (3.5) of the proposed solution.

3.1 System Requirements

The main architectural elements that the solution is required to incorporate in order to tackle Big Data Mining problems are:

- An Infrastructure Layer consisting of a cluster of (virtual or physical) computing nodes (AWS).
- A Distributed Storage Layer able to handle very large datasets (Hadoop HDFS).
- A Resource Management Layer to manage underlying resources and supervise their distribution to various applications (Hadoop YARN).
- A Batch Processing Layer to take advantage of the underlying computing nodes and execute tasks on them in parallel (Hadoop MapReduce and Spark).
- An integration mechanism between Weka and the Batch Processing Layer (Map and Reduce Wrappers around Weka classes).
- A User Interface.

The performance requirements of a viable Big Data Mining solution are:

- Linear strong and weak scaling on Big Data Mining tasks.
- Minimal data movements in the cluster during task execution.
- Minimal task-setup and termination overheads.
- Better performance than Mahout.

Section 3.2 presents the first four layers of the system's architecture. Section 3.3 analyses the integration mechanism and presents the execution model. Section 3.4 explains the performance benefits of replacing MapReduce with Spark in the Batch Processing Layer. Finally, Section 3.5 presents the evaluation strategy.
3.2 System Architecture

At the Infrastructure Layer, the nodes are provided by AWS. The AWS management console provides virtual Linux-based compute instances dynamically on demand.

The software stack is based on the latest stable version of Hadoop (2.2.0). The Storage Layer is based on HDFS and spans across the disks of the AWS instances. Partitioning, distribution and replication of datasets saved on HDFS are handled automatically by the HDFS libraries.

The Resource Management Layer of the cluster consists of YARN. YARN deploys a global ResourceManager which negotiates available resources (execution containers, one or more per node) with the ApplicationMaster of each application. The per-application ApplicationMaster requests resources from the ResourceManager and allocates them to the Mappers and the Reducers of the user application. This is a major development in Hadoop2. In earlier versions containers were statically allocated at the beginning of execution and occupied resources even when idle. This practice could enable a long running application to cause starvation to other applications.

MapReduce is Hadoop's Batch Processing Layer. It uses YARN to obtain execution containers for MapReduce jobs and HDFS as a data source. It is important to note that Hadoop's supported workloads do not limit to MapReduce and can host other frameworks as well. In the proposed system, MapReduce was used for testing the execution model before deploying the solution to Spark. The two frameworks co-exist in the cluster and MapReduce can continue to execute legacy applications.

Figure 3.1 depicts the system architecture:

![System Architecture Diagram](image-url)

*Figure 3.1: System Architecture.*
The Weka classes that implement Map and Reduce interfaces can be executed in parallel on the Hadoop cluster. The following section describes the integration mechanism and the execution model in detail.

3.3 Execution Model

The execution of Weka classes on top of Hadoop clusters is based on the packages released by the core development team of Weka [48].

The dataset is split into chunks that are triplicated and distributed to the disks of the cluster's nodes. Optionally, a dataset stratification step can be used to avoid skewed partial models. The ApplicationMaster allocates the containers granted by the ResourceManager to the Mappers to execute tasks. Mappers contain Weka classes wrapped by the Map interface. These classes access the local chunks on the nodes and learn a number (depends on the nodes of the cluster) of partial Weka models in parallel during the Map phase. In order to combine these models to form the Final Model during training, a single Reduce container takes all the partial models as input and executes a Meta-Learner against them in order to form a voted ensemble among them and produce the final output.

For example, assuming a dataset with N chunks of size S and a nominal class attribute, N Mappers are launched by the system to train N Decision Trees in parallel. After Mappers complete training, a Reducer receives N models as input and applies a Bagging predictor against them. The Final Model will consist of a function that outputs the majority class predicted by the N Decision Trees.

During model evaluation, the output model is replicated and distributed to Mappers to perform testing in parallel. During the Map phase, the model is tested against dataset instances and uses a combiner to perform local aggregation before submitting results to the Reducer. A single Reducer receives the locally aggregated results of all the Mappers, combines them, produces the global results and displays the output to the user.

Figure 3.2 provides a summary of the procedure.
This approach can naturally be used to build and test classifiers and clustering algorithms. While for classifiers Weka currently contains Meta-Learners (Bagging is the most important), there is currently no support for consensus (ensemble) clustering in the core packages. The WekaUT [49] extension provides an abstract class for consensus clusterers and can be used to provide a solid implementation. Text filtering and classification are also possible using the same process.

Note that neither the architecture nor the execution model are tied to a specific batch processing framework. The same model applies to both Hadoop's native MapReduce framework and Spark.

3.4 Spark and Main-Memory Caching

Hadoop's MapReduce has proven its value the last decade. However, recent developments in the hardware industry will force Hadoop-based solutions to make appropriate adjustments.

Main memory density continues to follow Moore's law [50]. As a result, Infrastructure Layers consisting of commodity servers with 128-256 GB of main memory are now commonplace. Clusters of commodity servers can reach many terabytes. Stoica [50] also notes that 90% of datasets handled by web giants Facebook, Yahoo and Bing are within that limit and would be benefit greatly performance-wise by main-memory caching. Currently, Hadoop's MapReduce only support data loading from HDFS.

Weka already supports main-memory caching for sequential tasks. However, this can only be used in a single Mapper instance, is not fault-tolerant and, most importantly, there is no mechanism to
go out of core if memory runs out (the job will fail). These features limit memory-intensive applications that require multiple Map waves. What is more, a main-memory abstraction at the framework level would enable interactive launch of many algorithms against a hot dataset.

This is the reasoning behind replacing MapReduce with Spark in the Batch Processing Layer in the final version of the system. Since both platforms are Java-based and implement the MapReduce paradigm switching between the solutions is a matter of context configuration.

The next section proceeds to present the evaluation strategy.

### 3.5 Evaluation

Two commonly used criteria when assessing parallel and distributed systems are ([51],[18]):

1. **Weak scaling**: the execution time variability of a growing problem with a fixed per-node size.
2. **Strong scaling**: the execution time variability of a growing system in a fixed size problem.

For weak scaling the aim is a constant execution time when increasing both problem size and the number of nodes. This is also called *scale-up* [51]. In order to measure this, the problem size will be doubled every time and tested against 2,4,8,16 and 32 nodes.

For strong scaling the aim is to have a linear relation between the cluster size and the execution time. This is also called *speed-up* [51]. In order to measure speed-up, a fixed size dataset will be processed by 2,4,8,16 and 32 nodes.

Another important parameter in evaluating the solution is measured resource utilisation. Network bandwidth can become a performance bottleneck [19] in large Hadoop clusters. It is therefore important to measure the network resource use during execution. These measurements will be produced using the Netperf [52] benchmark.

The performance of the system will be tested in three different configurations: Weka on MapReduce, Weka on Spark and Weka on Spark with limited main-memory. The project aims to achieve better performance than Mahout. Since the system uses main-memory caching, it will be interesting to observe behavior at the edge condition where main memory is exhausted and the underlying operating system takes action.

### 3.6 Summary

This section described the fundamental architectural components of the Hadoop cluster, the execution model of Weka based on the MapReduce paradigm, the reasoning behind selecting Spark as the Batch Processing Layer and the evaluation strategy of the implementation.

The next Chapter proceeds to describe progress made to date towards the implementation analysed above.

### 4. Progress

The following sections present the Hadoop environment setup (4.1), building and using Weka on top of Hadoop's MapReduce (4.2), preliminary test results (4.3), experiments on the Spark platform (4.4) and future planning (4.5).
4.1 Setting up the Hadoop Environment

The main actors of a Hadoop 2.2.0 cluster are:

1. the ResourceManager that performs the application level scheduling;
2. the ApplicationMaster that performs the task scheduling per application;
3. the NodeManager at each node that manages node resources;
4. the NameNode that holds the clusters metadata (locations of data chunks);
5. a SecondaryNameNode to avoid a single point of failure;
6. DataNodes that hold the data chunks in their local disks.

Depending on the amount of machines available these roles can be performed by different physical machines or can be combined in a single machine. Each node also holds a number of containers in which Hadoop tasks can be scheduled for execution. The proposed system was implemented in two different set-ups: a local testbed in a single virtual machine for testing and a distributed version using Amazon Elastic MapReduce on Amazon Web Services.

4.1.1 Local Testbed

The local testbed consists of a Linux-based virtual machine image. It contains the operating system, the essential Hadoop libraries, configuration files and the appropriate scripts that start and stop Hadoop daemons. In order to make the virtual Hadoop cluster accessible to client applications security and user accounts needed configuration. What is more, a Graphical User Interface (GUI) was installed.

Hadoop has two security options: SIMPLE and KERBEROS. The latter requires authenticating users using the Kerberos protocol [53]. As there is only one user at this stage by setting the security authentication property to SIMPLE Hadoop will just check if the remote account name matches the local account.

The next step is to create a new user, add him to the group of Hadoop users and grant him read, write and execute privileges in HDFS. This would grant him a directory in the HDFS and permissions to upload files and execute Map and Reduce jobs against them.

The virtual machine is hosted on a system with 4 physical cores and 8 logical (Intel Core i7-4700MQ processor). By changing the amount of cores that the guest system can use, it is possible to define the degree of parallelism in Hadoop jobs. For example, by assigning all 4 physical cores to the guest system, up to 8 Mappers can be executed in parallel. This feature is useful in testing scalability locally before deploying to AWS.

4.1.2 Amazon Web Services

The large scale environment used for testing the system is provided by Amazon's web service EMR (Elastic Map Reduce) [54]. It automatically installs Hadoop to a number of EC2 (Elastic Compute Cloud) [55] instances and provides a Hadoop cluster dynamically. EMR clusters are reconfigurable: nodes can be added or removed from the cluster without affecting running applications.

The system does not expose the Hadoop IPC (Inter-Process Communication) ports through the public DNS (Domain Name System) and job submission requires connection via a secure shell. A user account is automatically created, the account is granted the appropriate privileges and the system is ready to use.
4.2 Weka on Hadoop

Weka package manager already contains libraries that wrap Hadoop Mappers and Reducers around Weka's classifiers [48]. However, these packages are built against the first version of Hadoop that has become obsolete and does not support Spark integration. In order to transfer the solution to a Hadoop 2.2.0 cluster the packages must be rebuilt. This required editing the configuration files that contain the package dependencies on Hadoop libraries so they point to the Maven [56] repositories of the latest version. At this stage, by executing a build script the new packages are ready to install.

Installation requires accessing the host machine and using the command line tool of Weka's package manager. At this point, the system is ready for testing.

In order to execute mining tasks Weka offers a Graphical and a Command Line Interface (CLI). At remote servers a GUI is usually not present and CLI facilitates scripting.

Job submission requires configuring the following parameters in the script:

- Type of the task: Learning task, Evaluation or both
- Algorithm used in the Map phase
- Meta-learner for model combination in the Reduce phase
- Input and Output paths in HDFS
- Hadoop IPC ports
- Number of folds (for cross-validation)
- Chunk size
- Stratification (number of randomized chunks)

The last two are the most important since the amount of Mappers scheduled by the system (and thus the degree of parallelism) will depend either on the amount of chunks (if not stratified) or on the amount of randomized chunks.

4.3 Preliminary Test Results

The system was installed and tested both locally and in AWS. The tests perform training and evaluation tasks on classifiers. The following sections present the results.

4.3.1 On Local Testbed

Local Testbed results were mainly used to obtain insights on how various parameters affect the system.

4.3.1.1 Building Headers

Weka pays great attention to meta-data about the datasets. Consequently, the first step is to generate the header of the .arff file (Weka's supported file format). These headers contain a list of the attributes, their types and various statistics. After their generation, they reside in HDFS and each Map task obtains a copy before beginning execution. Additionally, this step adds the Weka libraries to the distributed cache of HDFS and initiates the logger.

When headers are built, Mappers compute statistics at dataset chunks and Reducers aggregate the results. The system will launch a Mapper for every chunk and the degree of parallelism will be
defined by the execution containers in the cluster. Figure 4.1 presents the results against a 25MBytes dataset. There is a flat overhead of 7.7s that was subtracted (starting Hadoop processes, setting-up classpath and logger).

<table>
<thead>
<tr>
<th>Chunks</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elapsed Time</td>
<td>6</td>
<td>6</td>
<td>4.1</td>
<td>3.9</td>
<td>6.4</td>
<td>8.4</td>
</tr>
</tbody>
</table>

The figure shows that the best performance is achieved when there is one Mapper for each physical core. The speed-up is not linear, mainly because there is overhead associated with launching the Mapper threads and because the host system reserves most resources for its own processes. This cost dominates as the amount of chunks increases. This parameter must carefully tuned to match the nodes in the cluster.

4.3.1.2 Linear Regression

The next test consists of training a Linear Regressor against a 500MB dataset. The Mappers were controlled this time by stratifying the dataset and setting the number of randomized chunks. Figure 4.2 depicts the results.
Figure 4.2: Classifier Training

Execution time improves as Mapper threads reach the maximum theoretic amount of threads that the host machine can handle (4 physical processors contain 8 logical processors and can handle 8 threads). The speed-up is not linear because the host system reserves resources needed for the two operating systems (host and guest). However, execution time decreases almost 50% when 4 Mappers are deployed indicating that doubling the execution containers lead to almost 100% speed-up.

Finally, Figure 4.3 illustrates the evaluation performance of the classifier built above and demonstrates similar behavior.
4.3.2 On AWS

The system was also tested on AWS. In order to build Weka on top of EMR a script to download and install the custom Hadoop2 packages from the Local Testbed's repository was implemented. This script is reusable for all Linux-based remote Hadoop clusters and it will be useful going forward.

The classifier training task was repeated in the larger environment on clusters of 4 and 8 nodes. Each node contains the AWS equivalent of a single-core 3GHz processor and 4GB of main-memory. Figure 4.4 presents the results.

<table>
<thead>
<tr>
<th>Randomized Chunks</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 Nodes</td>
<td>237</td>
<td>141</td>
<td>127</td>
<td>131</td>
</tr>
<tr>
<td>8 Nodes</td>
<td>179</td>
<td>117</td>
<td>79</td>
<td>82</td>
</tr>
</tbody>
</table>

![Classifier Training](image)

There is an overhead associated with dataset stratification and distribution to the nodes. This step requires shuffling and redistributing the data chunks between the nodes and thus consumes network resources. What is more, there is a single Reduce step for all cluster sizes. Since Reduce combines the produced models and does not access data the execution time is constant and independent of the dataset size. At large dataset that computations dominate execution the Reduce step is expected to be insignificant.

Apart from the issues above, the system demonstrates nearly linear strong scaling.
4.4 Spark

The local Testbed described in section 4.1.1 was used as the host system for both Hadoop and Spark. Spark installation required installing pre-compiled binaries to the virtual machine. The system was configured by pointing Spark execution engine to the communication ports of HDFS and YARN. After these steps, the system can launch Application Masters through YARN and share the cluster’s resources with existing MapReduce applications.

The system was tested by launching Scala-based Spark's shell, loading data to main memory in the form of RDDs and launching Map and Reduce jobs against them interactively. The most useful outcome going forward is the interoperability between MapReduce jobs and Spark's operators. Map and Reduce tasks written targeting the MapReduce framework can be launched inside Spark's Map and ReduceByKey operators. This was tested by using the classic WordCount example. Since all Map and Reduce tasks implement the same Mapper and Reducer interfaces, it is possible to use the Hadoop wrappers for Weka that were tested in the previous sections as arguments to Scala functions (both Java and Scala work on JVM (Java Virtual Machine) and can call each other's methods).

4.5 Plan

Table 4.1 depicts the post-examination implementation plan.

<table>
<thead>
<tr>
<th>Unique Id</th>
<th>Activity</th>
<th>Task Duration</th>
<th>Start Date</th>
<th>End Date</th>
<th>Percent completed</th>
<th>dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Progress Report Submission</td>
<td>1</td>
<td>9-May-2014</td>
<td>9-May-2014</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Semester 2 Finals</td>
<td>12</td>
<td>10-May-2014</td>
<td>21-May-2014</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>Spark Java API</td>
<td>8</td>
<td>22-May-2014</td>
<td>29-May-2014</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>Spark and Hadoop Context Configuration</td>
<td>9</td>
<td>30-May-2014</td>
<td>7-Jun-2014</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>Distributed Headers Implementation</td>
<td>10</td>
<td>8-Jun-2014</td>
<td>17-Jun-2014</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>Spark MapReduce Wrappers Implementation</td>
<td>16</td>
<td>18-Jun-2014</td>
<td>4-Jul-2014</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>Evaluation Jobs Implementation</td>
<td>8</td>
<td>5-Jul-2014</td>
<td>12-Jul-2014</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>Debugging on Local Testbed</td>
<td>8</td>
<td>13-Jul-2014</td>
<td>21-Jul-2014</td>
<td>0</td>
<td>3,4,5,6</td>
</tr>
<tr>
<td>9</td>
<td>Deployment at AWS and Evaluation</td>
<td>3</td>
<td>22-Jul-2014</td>
<td>24-Jul-2014</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>Results Interpretation</td>
<td>2</td>
<td>25-Jul-2014</td>
<td>27-Jul-2014</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>11</td>
<td>Final Report</td>
<td>39</td>
<td>28-Jul-2014</td>
<td>4-Sep-2014</td>
<td>0</td>
<td>all</td>
</tr>
<tr>
<td>12</td>
<td>Final Report Submission</td>
<td>1</td>
<td>5-Sep-2014</td>
<td>5-Sep-2014</td>
<td>0</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 4.1: Implementation Plan
5. Conclusion

This report presented a methodology to integrate Weka with scalable distributed computing frameworks that support the MapReduce paradigm. Preliminary test results on Hadoop demonstrate scalability and indicate that Weka's rich collection of algorithms can be combined with the power of modern clusters.

In ongoing work, efforts are made to integrate Weka with the emerging platform Spark. The aim is to take advantage of Spark's enhanced set of operators and main-memory caching and deliver considerable performance improvement as opposed to the existing system.

References


[41] K. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Eltabakh, C.-C. Kanne, F. Ozcan, and E. J. Shekita, 
“Jaql: A Scripting Language for Large Scale Semistructured Data Analysis,” 
In PVLDB, 2011.

[42] A. Ghoting , R. Krishnamurthy , E. Pednault , B. Reinwald , V. Sindhwani , S. Tatikonda , 
Y. Tian , S. Vaithyanathan, “SystemML: Declarative machine learning on MapReduce,”
in Proceedings of the 2011 IEEE 27th International Conference on Data Engineering, p.231-242, 
April 11-16, 2011

[43] M. Perez,A. Sanchez,A. Herrero,V. Robles, and Pea, Jos, M.,
“Adapting the Weka Data Mining Toolkit to a Grid Based Environment,” Advances in Web 
Intelligence, pp. 492–497, 2005.

Carleton College, CS TR (2002)

Mining on Grids,” in: 9th European Conference on Principles and Practice of Knowledge 
Discovery in Databases, Porto, Portugal, 2005.

“Toolkit-based high-performance data mining of large data on mapreduce clusters,”

“Mlbase: A distributed machine-learning system,” In Conf. on Innovative Data Systems 
Research, 2013.

[48] M. Hall, "Weka and Hadoop" blog, 15 October 2013; 
http://markahall.blogspot.co.uk/2013/10/weka-and-hadoop-part-1.html .


[50] I. Stoica, "Berkeley Data Analytics Stack (BDAS) Overview;"

[51] "Measuring Parallel Scaling Performance," 
https://www.sharcnet.ca/help/index.php/Measuring_Parallel_Scaling_Performance  


