A Parallel Distributed Weka Framework for Big Data Mining using Spark

Aris-Kyriakos Koliopoulos, Paraskevas Yiapanis, Firat Tekiner, Goran Nenadic, John Keane
School of Computer Science, The University of Manchester, Manchester, UK
Email: aris.koliopoulos@postgrad.manchester.ac.uk,{paraskevas.yiapanis, firat.tekiner, gnenadic, john.keane}@manchester.ac.uk

***Abstract*—Effective Big Data Mining requires scalable and efficient solutions that are also accessible to users of all levels of expertise. Despite this, many current efforts to provide effective knowledge extraction via large-scale Big Data Mining tools focus more on performance than on use and tuning which are complex problems even for experts.**

Weka is a popular and comprehensive Data Mining workbench with a well-known and intuitive interface; nonetheless it supports only sequential single-node execution. Hence, the size of the datasets and processing tasks that Weka can handle within its existing environment is limited both by the amount of memory in a single node and by sequential execution.

This work discusses `DistributedWekaSpark`, a distributed framework for Weka which maintains its existing user interface. The framework is implemented on top of Spark, a Hadoop-related distributed framework with fast in-memory processing capabilities and support for iterative computations.

By combining Weka's usability and Spark's processing power, `DistributedWekaSpark` provides a usable prototype distributed Big Data Mining workbench that achieves near-linear scaling in executing various real-world scale workloads - 91.4% weak scaling efficiency on average and up to 4x faster on average than Hadoop.

***Keywords*-Weka; Spark; Distributed Systems; Data Mining; Big Data; Machine Learning**

I. Introduction

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" [1]. In practice, the term refers to datasets that are increasingly difficult to collect, curate and *process* using traditional methodologies. Effective Big Data Mining requires scalable and efficient solutions that are also easily accessible to users at all levels of expertise.

Distributed systems provide an infrastructure that can enable efficient and scalable Big Data Mining. Such systems, made up of organized clusters of commodity hardware, process large volumes of data in a distributed fashion. Hadoop [2], an open source implementation of MapReduce [3], is the most widely-used platform for large-scale distributed data processing. Hadoop processes data from disk which makes it inefficient for data mining applications that often require iteration. Spark [4] is a more recent distributed framework that works with Hadoop and provides in-memory computation that allows iterative jobs to be processed much faster, hence making it a more suitable base for data mining.

A difficulty in developing large-scale data mining toolkits is how to express the algorithms in such a way to make them as easy to use as existing sequential tools [5]. Most attempts expose a restricted set of low-level primitives such as MapReduce but usually tend to be prohibitive due to their complex nature and inability to accommodate the patterns of data mining algorithms. Data analysts aim to extract knowledge and to better understand their data; they do not wish to learn complex programming paradigms and new languages. Recent implementations attempt to provide high-level interfaces for data mining and associated algorithms which are compiled to low-level primitives [6], [7]. Such developments tend to require knowledge of the underlying distributed system effectively shifting the focus from data mining to individual algorithm implementation.

Weka [8] is a widely used data mining tool [9] that supports all phases of the mining process, encapsulates well tested implementations of many popular mining methods, offers a GUI that supports interactive mining and result visualization, and automatically produces statistics to assist result evaluation.

However, a major disadvantage of Weka is that it only supports sequential single-node execution and hence has significant limitations in handling Big Data [10].

To address the need for efficient and scalable processing, allied to ease-of-use and seamless transformation between platforms for Big Data Mining, we have designed and prototyped `DistributedWekaSpark` to leverage Weka and distributed systems via Spark.

This paper makes the following contributions, it:

- develops a cloud-ready system, easily accessible to users of all levels of expertise, that allows efficient analysis of large-scale datasets. Users already familiar with Weka can seamlessly use `DistributedWekaSpark`;
- extends the existing Weka framework without the need to re-implement algorithms from scratch. This enables faster porting of existing systems and allows existing users to use the same interface for both local and distributed data analysis;
- describes a unified framework for expressing Weka's algorithms in a MapReduce model. This eliminates the need to inspect algorithms to identify parallel parts and re-implement them using MapReduce;

- evaluates `DistributedWekaSpark`'s performance on various real-world scale workloads - the system achieves near-linear scaling and outperforms Hadoop by a factor of 4 on average on identical loads.

II. Distributed Frameworks: MapReduce, Hadoop, Spark

Google in 2004 [3] introduced MapReduce, a distributed computing model targeting large-scale processing. MapReduce expresses computations using two operators (Map and Reduce), schedules their execution in parallel on dataset partitions and guarantees fault-tolerance through replication. *Map* processes dataset partitions in parallel; *Reduce* aggregates the results. A MapReduce distributed system consists of a Master node which handles data partitioning and schedules tasks automatically on an arbitrary number of Workers. Once the functions are specified, the runtime environment automatically schedules execution of Mappers on idle nodes. Each node executes a Map function against its local dataset partition, writes intermediate results to its local disk and periodically notifies the Master of progress. As the Mappers produce intermediate results, the Master node assigns Reduce tasks to idle nodes.

Yahoo in 2005 introduced Hadoop [2], an open source implementation of MapReduce. The Hadoop Distributed File System (HDFS) is a disk-based file system that spans across the nodes of a distributed system. Files stored in HDFS are automatically divided into blocks, replicated and distributed to the nodes' local disks. HDFS maintains metadata about the location of blocks and assists Hadoop to schedule each node to process local blocks rather than receive remote blocks through the network. HDFS encapsulates distributed local storage into a single logical unit and automates the procedure of distributed storage management.

Although MapReduce can express many data mining algorithms efficiently, significant performance improvement is possible by introducing a loop-aware scheduler and main-memory caching. Data mining algorithms tend to involve multiple iterations over a dataset and thus, multiple, slow, disk accesses. As a consequence, storing and retaining datasets in-memory and scheduling successive iterations to the same nodes should yield significant benefits. Modern nodes can use performance-enhancing main-memory caching mechanisms.

Spark [4] supports main-memory caching and possesses a loop-aware scheduler. Additionally, Spark implements the MapReduce paradigm and is Java-based (as is Hadoop). These features enable users to deploy existing Hadoop application logic in Spark via its Scala API. Spark has been shown to outperform Hadoop by up to two orders of magnitude in many cases [11].

MapReduce was designed to develop batch applications that process large amount of data using a large number of nodes. During this process disk is used as the intermediate storage medium (as opposed to memory) to share data amongst different stages and iterations of an application. In contrast, Spark enables a near-real time application development framework with more effective use of memory for different stages of an application to communicate and share information. This allows Spark to overlap I/O and computation more efficiently as, without waiting for a partition to be fetched from HDFS, multiple partitions can be loaded at once. Spark is fully compatible with Java-based applications as it uses Scala as its programming language which runs a Java Virtual Machine, hence allowing easier integration. Further, it allows the user to use anonymous functions which enables chaining multiple computation in to a single line and leveraging a larger number of functional operators.

Spark provides Resilient Distributed Datasets (RDDs) [11], a distributed main-memory abstraction, that enable users to perform in-memory computations in large systems. RDDs are an immutable collection of records distributed across the system's main memory. These data structures can be created by invoking a set of operators either on persistent storage data objects or on other RDDs. RDD operators are divided into two categories: *Transformations* and *Actions*. Transformations define a new RDD, based on an existing RDD and a function; actions materialize the transformations and either return a value to the user or export data to persistent storage.

Transformations provide native support for MapReduce. When users execute tasks, the system creates RDDs (from the input) and distributes their records across its main memory. When operations are issued each node processes its local set of records and return results. Caching datasets to main memory avoids slow disks reads and performs much faster than Hadoop's MapReduce on algorithms such as Logistic Regression and K-means [11].

When the dataset is larger than the main memory, Spark employs a mechanism to serialize and store portions of the dataset to secondary storage. The system offers numerous memory management options including different serialization and compression libraries. These features allow the user to define an application-specific caching strategy that takes advantage of both dataset and system characteristics.

III. DistributedWekaSpark: A Big Data Mining Architecture

This section demonstrates how `DistributedWekaSpark` fits in the context of a Big Data Mining process. Figure 1 shows the various components of an efficient and scalable solution for Big Data Mining; Figure 2 presents a layered view of the system's architecture.

The architecture includes the following components: an *Infrastructure* layer consisting of a reconfigurable cluster of either physical or virtual computing instances, e.g. using Amazon EC2 instances; a *Distributed Storage* layer

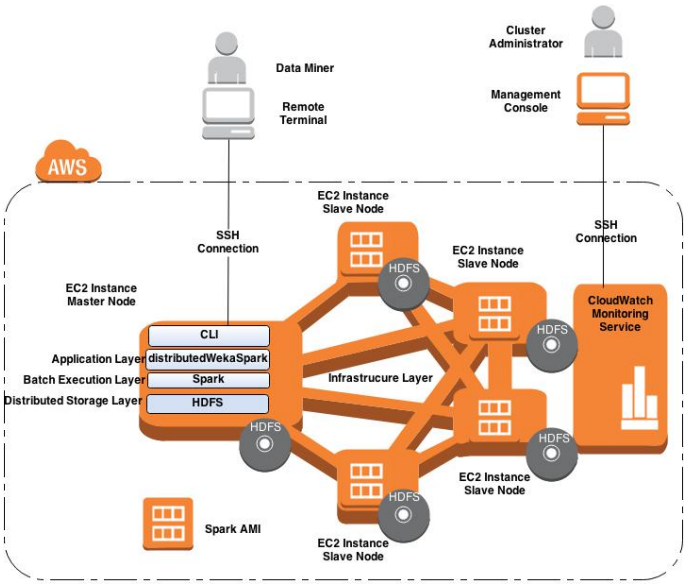

Figure 1. Components of a Big Data Mining process

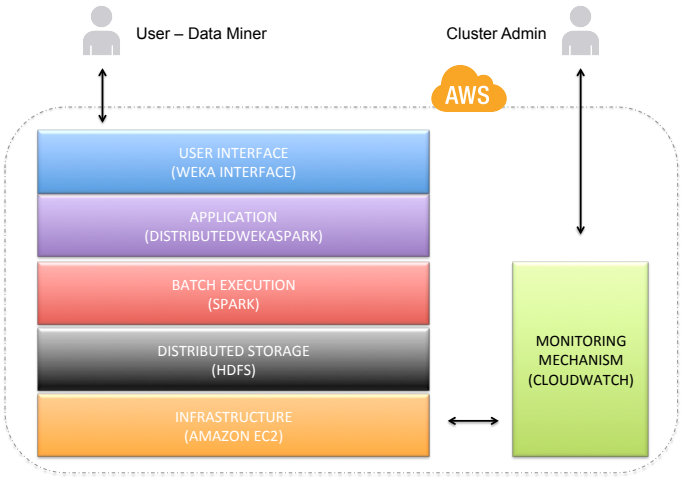

Figure 2. Layered view of a Big Data Mining architecture

which automatically encapsulates the local storage of the system instances into a large-scale logical unit, e.g. HDFS; a *Batch Execution* layer which schedules and executes tasks on data stored in distributed storage, e.g. Spark; an *Application* layer that integrates the application logic of Big Data Mining workloads into the programming model supported by the Batch Processing layer - this is where `DistributedWekaSpark` is used; a *User Interface* for interaction with the system to run Data Mining tasks; and a *Monitoring Mechanism*, used for performance tuning and system evaluation (e.g. CloudWatch). This structure is also used in [11].

`DistributedWekaSpark` is cloud-ready and can be accessed through an interface that mimics Weka's simple Command Line Interface. For example, a classifier can be built with a simple command such as:

```
>bin/spark DistributedWekaSpark.jar -path
<dataset> -task buildClassifier
```

Default values are provided for Spark and Weka options with the user allowed to change preference, e.g. choice of a specific classifier and the number of folds. Hence, users familiar with Weka can use `DistributedWekaSpark` "out of the box".

IV. Design of DistributedWekaSpark

This section describes `DistributedWekaSpark`'s design, implementation, and execution on Spark. We explain how `DistributedWekaSpark` enables Data Mining methods [12] to be ported for distributed processing. We also provide examples of how methods such as Classification, Regression and Association Rules have been extended to run on Spark. Note that the system user is unaware of the implementation details discussed in this section. We first consider how sequential Weka model works.

A. The Weka Model

Sequential Weka allows easily new implementations of Data Mining methods to be added by taking advantage of Object Oriented design principles [13]. The framework provides abstract interfaces for developers to build and evaluate such models. For example, to develop a classification algorithm Weka provides the *Classifier* interface which specifies two abstract methods, one to build a classifier and one to classify the instances using the classifier previously built. The developer inherits these two methods for their algorithm and provides their own implementations. Every algorithm for classification redefines the interface according to how it builds a classifier and how it classifies instances. This allows a uniform interface for building and using classifiers from other code.

In a similar way, Weka provides interfaces for Association Rules and Clustering in order to provide developers a uniform way to build and evaluate their models.

Further, when Weka first reads the data, it generates a special object to structure the dataset in such a way that data can be easily and quickly accessible later on during model creation. Alongside, another structure called *Headers* is created to hold useful statistics for each dataset. More details are provided in Section IV-C.

B. Distributed Model Requirements

Extending Weka for distributed execution on Spark requires to express the following steps using MapReduce functions:

- RDD generation from raw data;
- Header creation using the RDDs;
- Model creation;
- Model evaluation;

The following sections describe `DistributedWekaSpark`'s approach to each of these steps.

Our unified framework focuses on the steps for creating and evaluating models where various implementations may exist. Note that RDDs and headers are created only once and then reused across executions regardless of the Data Mining model.

C. RDD Generation

Upon submitting a task using Weka's interface and a set of parameters, the application's main thread is invoked in the Spark Master. The application parses user options using a custom text parser, configures the essential environment parameters (application name, total number of cores, per-instance cache memory, etc.) and initializes a Task Executor. The Task Executor begins the execution procedure by defining an RDD (raw data) from the data on HDFS.

RDD Transformations are "lazy": until an Action is issued (a Reduce operator in this case), the RDD will not be materialized. Thus, the RDD at this stage is logical.

Weka processes data in a special-purpose object format known as *Instances*. This object contains a *header* (meta-data about the attributes) and an array of *Instance* objects. Each Instance object contains a set of attributes which represents the raw data. HDFS data on Spark are usually defined as RDDs of Java String objects. Thus, a Transformation is needed to parse the strings and build an Instances object for each partition. This is achieved by defining a new RDD (dataset), based on the previous RDD (raw data) and a Map function. These Transformations are logged by Spark into a lineage graph. The Task Executor will then use the newly defined RDD as an initialization parameter for the user requested task. These tasks will add their own Transformations to the graph. When an Action is issued (a Reduce function), Spark schedules node instances to build the RDD partitions from their local dataset partitions, and to materialize the Transformations in parallel.

D. Headers Phase

Weka pays particular attention to metadata. An essential initial step is to compute the header of the ARFF file (Weka's supported format, represented by the aforementioned Instances object at runtime). A header contains attribute names, types and multiple statistics including minimum and maximum values, average values and class distributions of nominal attributes. Figure 3 displays the MapReduce job that computes the dataset's header file.

```
headers=rawData
        .map(new CSVToArffHeaderSparkMapper(options,names,numOfAttributes).map(_))
        .reduce(new CSVToArffHeaderSparkReducer().reduce(_,_))
```

Figure 3. Header creation MapReduce job

The job requires the attributes names, the total number of attributes and a set of options. These parameters are used by the Map function to define the expected structure of the dataset. Map functions compute partition statistics in parallel; Reduce functions receive input from the Map phase and aggregate partition statistics into global statistics.

This procedure is only mandatory for nominal values, but can be invoked for any type of attributes. Upon creation, Headers are distributed to the next MapReduce stages as an initialization parameter. This procedure is required *only once* for each dataset; Headers can be stored in HDFS and retrieved upon request.

E. A Unified Framework Approach

The high-level primary goals of data mining are *description* and *prediction* and can be achieved using a variety of data mining methods [12]. Each data mining method has a number of possible algorithms; for instance, Classification includes Support Vector Machines (SVM), Decision Trees, Naive Bayes, etc. Each algorithm can in turn have different implementations. There are many sequential implementations for these algorithms and a relatively small number of methods, thus, our focus is on parallelization of methods themselves rather than individual implementations.

As discussed, previous attempts to introduce parallelism to Weka in a distributed environment [26], [27], [28] have reviewed the Weka libraries to identify algorithm parts to execute in parallel and then re-implemented using MapReduce. This does not provide a unified framework for expressing Weka's algorithms, rather it entails producing custom distributed implementations of each algorithm. This is complex and time-consuming, and the quality of the provided solutions varies significantly based on the contributor expertise. Lack of a unified execution model leads to inconsistent performance, difficulties in maintaining and extending the codebase and discourages widespread adoption. Thus, here we have focused on building execution models for Data Mining methods rather than providing implementations of specific algorithms.

To enable a unified execution model, we exploit the Object-Oriented properties of the Weka implementation. Weka represents each method of Data Mining algorithms (e.g. classification, regression, etc.) using an abstract interface. Individual algorithms must implement this interface. By implementing Map and Reduce execution containers ("wrapper") for Weka's interfaces, a scalable execution model becomes feasible. This enables all the algorithms currently implemented for sequential Weka to utilize distributed processing. `DistributedWekaSpark`'s interface mimics that of Weka, hence users can design and execute Data Mining processes using the same tools either locally or in distributed environments. The proposed execution model is based on a set of packages released by the Weka development team [14], extended in this work to Spark's API and Scala's functional characteristics.

F. Building & Evaluating Models

Spark begins execution by scheduling the slave instances to load local dataset partitions to main memory. Each slave invokes a unary Map function containing a Weka algorithm against a local partition and learns an intermediate Weka model. Intermediate models generated in parallel are aggregated by a Reduce function and the final output is produced. However, the order of operands in the Reduce functions is not guaranteed. Consequently, Reduce functions have been carefully designed to be associative and commutative, so that the arbitrary tree of Reducers can be correctly computed in parallel.

The functional model demands stateless functions. Spark provides a mechanism to broadcast variables, but this practice introduces complexity, race conditions and network overheads. As a result, Map and Reduce functions have been designed to solely depend on their inputs. As Map outputs consist of Weka models (plain Java objects), this should minimize network communication between the nodes during execution.

1) Classification and Regression: Classifiers and Regressors are used to build prediction models on nominal and numeric values respectively. Although many learning algorithms in these categories are iterative, both training and testing phases can be completed in a single step using asynchronous parallelism. It is important to emphasize the performance improvement offered by Spark in multi-phase execution plans. Once the dataset is loaded into main memory in the Header creation phase, Spark maintains a cached copy of the dataset until explicitly told to discard. This feature offers significant speedup in consecutive MapReduce phases, as redundant HDFS accesses required by Hadoop are avoided.

Model Training: Once the Headers are either computed or loaded from persistent storage, Spark schedules slaves instances to begin the training phase. Every instance possesses a number of cached partitions and trains a Weka model against each partition, using a Map function. Classifiers and Regressors are represented in Weka by the same abstract object. Figure 4 displays the implementation of the model-training Map function.

By using Meta-Learning techniques, the intermediate models are aggregated using a Reduce function to a final model. Depending on the characteristics of the trained model the final output may be:

- a single model, in case the intermediate models can be aggregated (where a model of the same type as the inputs can be produced);
- an Ensemble of models, in case intermediate models cannot be aggregated.

```
class WekaClassifierSparkMapper(classifier:String,classifierOptions:Array[String]){

  val classifier=Class.forName(classifier).newInstance().asInstanceOf(weka.classifiers.Classifier)
  classifier.setOptions(classifierOptions)

  def map(partition:Instances): Classifier={
    classifier.buildClassifier(partition)
  return classifier
  }
}
```

Figure 4. Classification/Regression: Model-Training Map Function

Figure 5 displays the implementation of the model-aggregation Reduce function.

```
class WekaClassifierSparkReducer {

  reduce(classifierA:Classifier,classifierB:Classifier):Classifier={
    var aggregator:Classifier=null

    if(classifierA.isInstanceOf[Aggregateable]){
      aggregator.asInstanceOf[classifierA.getClass]
      aggregator.aggregate(classifierA)
      aggregator.aggregate(classifierB)
    }
    else{
      aggregator=new Vote
      aggregator.aggregate(classifierA)
      aggregator.aggregate(classifierB)
    }
  return aggregator
  }
}
```

Figure 5. Classification/Regression: Model-Aggregation Reduce Function

The trained models can be either used directly for testing unknown data objects or can be stored in HDFS for future use.

Model Testing and Evaluation: Once a trained model is either computed or retrieved from persistent storage, the Model-Evaluation phase can be completed in a single MapReduce step. The trained model is distributed to the slave instances as an initialization parameter to the Evaluation Map functions. During the Map phase, each instance evaluates the model against its local partitions and produces the intermediate evaluation statistics. Figure 6 displays the model Evaluation Map function.

```
class WekaClassifierEvaluationSparkMapper(classifier:Classifier,options:Array[String]){

  val evaluation=new Evaluation
  evaluation.setOptions(options)

  def map(partition:Instances): Evaluation={
    evaluation.evaluateModel(classifier,partition)
  return evaluation
  }
}
```

Figure 6. Classification/Regression: Evaluation Map Function

Reduce functions produce the final output by aggregating intermediate results. Figure 7 displays the implementation of the evaluation Reduce function.

In a similar fashion, trained models can be used to classify unknown instances.

2) Association Rules: Association Rules (ARs) are computed in parallel using a custom MapReduce implementation of the

```
class WekaClassifierEvaluationSparkReducer () {

  def reduce(evalA:Evaluation,evalB:Evaluation): Evaluation={
    val evaluation=new AggregateableEvaluation
    evaluation.aggregate(evalA)
    evaluation.aggregate(evalB)
  return evaluation
  }

}
```

Figure 7. Classification/Regression: Evaluation Reduce Function

Partition algorithm [15]. Partition requires two distinct phases to compute ARs on distributed datasets (not shown here due to space limitations). In the candidate generation phase, a number of candidate rules are generated from each partition. In the candidate validation phase, global support and significance metrics are computed for all the candidates and those that do not meet global criteria are pruned.

The user defines a support threshold and an optional threshold to any Weka-supported measure of significance (confidence is used by default). A number of Map functions proceed to mine partitions in parallel using a Weka AR learner and generate candidate rules. A rule is considered a candidate, if global significance criteria are met in any of the partitions. Candidate rules are exported from Map functions using a hash-table. Reduce functions aggregate multiple hash-tables and produce a final set of candidates. The hash-table data structure was selected because it enables almost constant seek time. In the validation phase, each Map function receives the set of candidates and computes support metrics for every rule. The validation Reduce phase uses the same Reduce function to aggregate the metrics across all partitions. Each rule that fails to meet the global criteria is pruned. The rules are sorted on the requested metrics and returned to the user.

V. Evaluation

A. Experimental Setup

The implementation has been evaluated using Amazon EC2 instances of three distinct configurations:

- **Small-scale**: an 8-core system of m3.xlarge instances possessing 28.2GB of main-memory;
- **Medium-scale**: a 32-core system of m3.xlarge instances possessing 112.8GB of main-memory;
- **Large-scale**: a 128-core system of m3.xlarge instances possessing 451.2GB of main-memory.

The dataset scales used in the evaluation process are proportional to the system size: **Small-scale**: 5GB; **Medium-scale**: 20GB; **Large-scale**: 80GB. The scale of data volumes used for evaluation was based on Ananthanarayanan et al. [16], which identified, based on analysis of access patterns of data mining tasks at Facebook, that 96% of tasks processed data that could be stored in only a fraction of the cluster's total main memory (assuming 32GB of main memory per server). Further, similar work by Appuswamy et al. [17] reports that the majority of real-world data mining tasks process less than 100GB of input.

`DistributedWekaSpark` includes three Data Mining methods: classification, regression and association rules. Each method has been evaluated by using a representative algorithm: classification using SVM, regression using Linear Regression, association rules using FP-Growth. In each case, the experiment was repeated

using the three scales. An identical workload was executed on Hadoop for comparison.

In terms of the evaluation, Hadoop was consistent across all three algorithms. Thus, due to space limitations, we show only Classification (SVM in this case) in terms of scaling efficiency and execution times against Hadoop. Classification was chosen as it underpins many of the fundamental algorithms in data science [18]. Further, we note that classification algorithms have relatively low levels of dependencies amongst stages which is challenging as it complicates memory utilization (compared to association rule mining).

B. Experimental Results

The system is assessed using two scalability metrics:

- **Strong Scaling**: the total problem size remains constant as additional instances are assigned to speed-up computations;
- **Weak Scaling**: the per-instance problem size remains constant as additional instances are used to tackle a bigger problem.

Figure 8a-c demonstrate `DistributedWekaSpark`'s strong scaling on three algorithms.

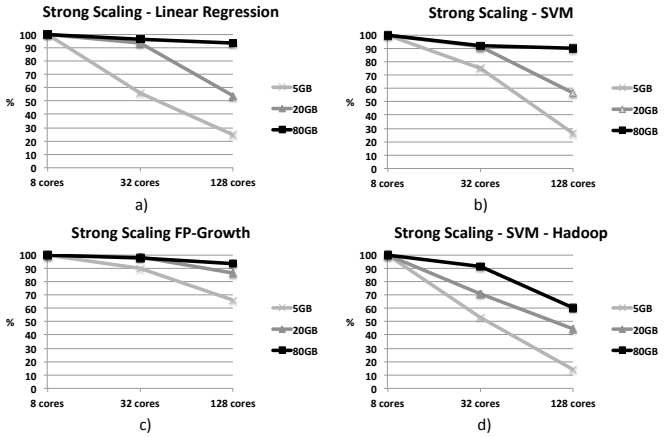

Figure 8. Strong scaling for a) Linear Regression, b) SVM, c) FP-Growth, and d) SVM run on Hadoop

Strong scaling efficiencies on Spark approach linearity when datasets are large and runtime is dominated by computation. Using large systems for small loads is inefficient due to initialization overheads. These overheads were measured at 11 seconds, regardless of the system's scale. On the largest system (128 cores), this number corresponds to 40.8% of the average total execution time on the small scale dataset (5GB), to 20.3% on the medium scale dataset (20GB), and to 10.1% on the large scale dataset (80GB). As the dataset size increases, runtime is dominated by computation and thus, overheads are amortized across total execution time. For comparison purposes, Figure 8d illustrates the strong scaling of Hadoop on SVM (vs. Figure 8b). Hadoop's strong scaling efficiency demonstrates inferior performance due to larger initialization overheads - Hadoop's initialization cost was 23 seconds. This overhead occurs at the beginning of every MapReduce stage, whereas in Spark it is only required at the first stage.

Figure 9 demonstrates the weak scaling efficiency of the three algorithms used for benchmarking. The figure also presents the weak scaling efficiency of the SVM algorithm on Hadoop.

Execution times approach linear performance for systems up to 128 cores in all cases. In contrast, Hadoop's weak scaling efficiency rapidly decreases as the number of cores increases.

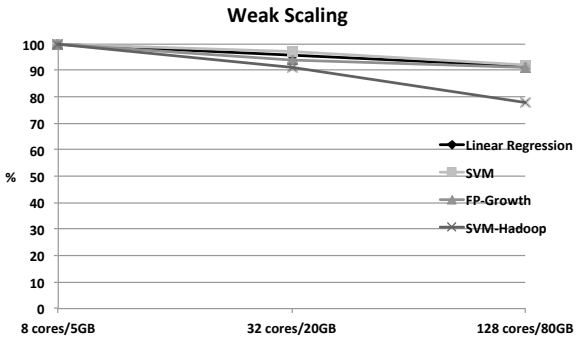

Figure 9. Weak scaling for Linear Regression, SVM, and FP-Growth on `DistributedWekaSpark` and SVM on Hadoop

In general, a slight performance decline is expected in fully distributed systems through monitoring multiple instances and load balancing. However, as Spark achieves high locality in consecutive stages, the effects are minimal in multi-stage execution plans.

Table I displays the average speedup of `DistributedWekaSpark` compared to Hadoop for building a classifier. Hadoop is used as the baseline, so any number greater than 1 means that `DistributedWekaSpark` is faster. Shaded cells indicate cases where full dataset caching was possible; for the rest of the cases only partial dataset caching was achieved. Hadoop loads a partition for each active Map container into memory, executes the Map task, discards the processed partition and repeats the procedure until the whole dataset is processed. In contrast, `DistributedWekaSpark` loads partitions until memory saturation and schedules the Map tasks to process in-memory data. Thus, when the available memory is larger than the dataset, `DistributedWekaSpark`'s caching of the dataset has obvious benefits (up to nearly 4x faster). Successive stages process the same RDDs so the need to reload and rebuild the dataset in the required format is avoided.

SPEEDUP	5GB	20GB	80GB
8 cores	2.13	1.78	1.69
32 cores	3.00	2.28	1.68
128 cores	3.97	2.23	2.52

Table I
AVERAGE SPEEDUP OF DISTRIBUTEDWEKASPARK AGAINST HADOOP FOR CLASSIFICATION.

In cases where the dataset cannot be fully cached, Spark applies a partition replacement policy where the Least Recently Used (LRU) partition is replaced. This process indicates that it is highly unlikely that successive stages will find the required partitions in-memory. Thus, the partition will be loaded from disk as in Hadoop. However, there is a big difference between the mechanisms Hadoop and Spark use to implement this process.

Hadoop reads HDFS partitions using an iterator. Map tasks read a HDFS partition line-by-line (each line being represented as a key-value pair), process each line and emit intermediate key-value pairs if necessary. In the specific case of Hadoop, the partitions are read line-by-line, each line is processed by a parser and then added to an Instances object (Weka's dataset representation). When this procedure is completed, the Map tasks execute the SVM algorithm and iterate over the data until the algorithm converges. When the

model is built, Hadoop emits the trained model, discards the data and schedules the Mapper to process a new partition. Thus, reading data from HDFS is coupled with data processing: while the system is reading data, the CPU is idle and while the system is processing data the I/O subsystem is idle. This process leads to suboptimal resource utilization: CPU cycles are wasted and I/O bandwidth is never saturated.

Spark resolves this issue using RDDs. This abstraction decouples the two phases. Map tasks process RDD partitions that are already in-memory. As the system is not required to wait for I/O and reads directly from main memory, maximum CPU utilization is achieved. Additionally, Spark removes older partitions from the distributed cache and fetches the next set of partition from HDFS regardless of the task execution phase. This gives faster reading of data as it is performed at a block rather than a line level, and overlaps data loading and data processing. These two features, along with the shortened initialization time, contribute to significant speedup compared to the Hadoop-based solution.

For example, in the smallest cluster configuration of 8 cores with ∼28GB main memory (see Table I), using 80GB of raw data produced 200GB of data in memory (we explain this behavior below). Spark was able to cache only 10% of the dataset effectively forcing disk reads; given the LRU policy, no suitable partition was available, yet Spark was still 1.69x faster on average than Hadoop.

Spark RDDs can be represented in memory as distributed Java objects. These objects are very fast to access and process, but may consume up to 5 times more memory than the raw data of their attributes. This overhead can be attributed to the metadata that Java stores alongside the objects and the memory consumed by the object's internal pointers. Spark offers a series of tools to tackle these overheads by introducing serialization and compression. In the experiments both serialization and compression were found beneficial as they demonstrated memory footprints close to the on-disk dataset values and reduction in main memory utilization by 40% at a small performance penalty (∼5% on average of the total execution time in our case). The Kryo serialization library was used as it offers better compression ratios than the built-in Java serialization [4].

VI. Related Work

Mahout [19], a community-based Hadoop-related project, aims to provide scalable data mining libraries. As its libraries do not provide a general framework for building algorithms, the quality of the provided solutions varies significantly being dependent on contributor expertise leading to potential lower performance [20]. Mahout also focuses on implementing specific algorithms, rather than building execution models for algorithm methods.

Radoop [21] introduced the RapidMiner toolkit to Hadoop. RapidMiner has a GUI to design work-flows which consist of loading, cleaning, mining and visualization tasks. Radoop introduced operators to read data from HDFS and execute Data Mining tasks. Its operators correspond to Mahout algorithms. At runtime the workflow is translated to Mahout tasks and executed on Hadoop. Radoop suffers the same performance issues as Mahout. However, it does separate Data Mining work-flow design from distributed computations.

Ricardo [22] merged the data mining/statistics tool R with distributed frameworks. The system uses a declarative scripting language and Hadoop to execute R programs in parallel. Although the system uses R-syntax familiar to many analysts, Ricardo suffers from long execution times due to the overhead produced by compiling the declarative scripts to low-level MapReduce jobs.

SparkR [6] provides R users with a lightweight front-end to a Spark system. It enables the generation and transformation of RDDs through an R shell. RDDs are exposed as distributed lists through the R interface. Existing R packages can be executed in parallel on partitioned datasets by serializing closures and distributing R computations to the nodes. Global variables are automatically captured, replicated and distributed to the system enabling efficient parallel execution. However, the system requires knowledge of statistical algorithms as well as basic knowledge of RDD manipulation techniques.

RHIPE [23] and SystemML [7] also aim to extend R for large-scale distributed computations. All the cited efforts to merge R with MapReduce suffer from two issues: (1) R is based on C and hence not native to Java-based frameworks such as Hadoop and Spark. Thus a bridging mechanism is required between R and the underlying Java Virtual Machine. R-code is compiled to C-code which uses the Java Native Interface for execution which reduces portability; (2) The underlying MapReduce paradigm is visible to the user. The user needs to express computations as a series of transformations (Map) and aggregations (Reduce). In Ricardo, this is achieved by using Jaql declarative queries where the selection predicates use R functions to transform the data (Map equivalent) and aggregation functions (Reduce equivalent) to produce the final output. SparkR uses the same methodology on distributed lists. RHIPE requires the user to learn MapReduce. SystemML allows experts to develop efficient distributed machine learning algorithms but requires programming in custom Domain Specific Language.

RABID [24] also allows R users to scale their work on distributed systems, particularly Spark. Although RABID provides an interface familiar to existing R users, the system still suffers from the performance and portability issues explained above.

These observations further support the decision to use Weka as it is written in Java and the bridging overheads of these systems are avoided. Additionally, by using Weka's interface, distributed MapReduce computations can be abstracted from the design of Data Mining processes.

Efforts to combine Weka's user model with the power of distributed systems include WekaG [26], WekaParallel [27] and Weka4WS [28]. WekaG and Weka4WS use web services to submit and execute tasks to remote servers. However, they do not support parallelism; each server executes an independent task on its own local data. WekaParallel proposed a parallel cross-validation scheme where each server receives a dataset copy, computes a fold and sends back the results. This practice cannot be applied on a large scale because of network bottlenecks. Work by Wegener et al. [10] aimed to merge Weka with Hadoop. Their methodology does not provide a unified framework for expressing Weka's algorithms. Each algorithm must be inspected to identify parts to execute in parallel and then re-implemented using MapReduce. This process entails producing custom-distributed implementations of all the algorithms in Weka and suffers from the same shortcomings as Mahout. Additionally, reading incrementally from disk produces large overheads on iterative algorithms.

VII. Conclusions and Future Work

`DistributedWekaSpark`[1] is a scalable Big Data Mining toolkit that exploits the power of distributed systems whilst retaining the standard Weka interface. `DistributedWekaSpark` is built on top of Spark which provides fast in-memory iterative processing, utilizing both parallel and distributed execution, making it ideal for Data Mining algorithms. Spark can be installed on Amazon EC2 systems by running a simple script and hence `DistributedWekaSpark` is ready to use with simple commands that mimic Weka's existing user interface.

[1]The code for `DistributedWekaSpark` can be found at *https://github.com/ariskk/distributedWekaSpark*

Evaluation shows that DistributedWekaSpark achieves near-linear scaling on various real-world scale workloads and shows speedups of up to 4x faster on average than Hadoop on identical workloads.

Further work will consider three areas: the provision of clustering, detailed analysis of caching performance and comparison with MLlib [25].

The execution model, that encapsulates Weka algorithms in MapReduce wrappers, is in theory applicable to any clustering approach. For example, Canopy Clustering [29] which divides the dataset into overlapping regions using a cheap distance function, can be implemented. Canopies within a threshold are assumed to represent the same region and hence can be aggregated: Map functions can be used to build Canopy Clusterers on partitions in parallel and Reduce functions can aggregate Canopies on the same region. However, this method will not work for other clustering approaches where aggregation would require the use of consensus clustering, as it is not yet supported by Weka.

A preliminary analysis of Spark's default caching strategy has shown it to be inefficient. As a result, multiple caching strategies have been investigated for data mining algorithms suggesting that: (1) serialization and compression mechanisms significantly decrease memory footprint with marginal performance overheads; (2) a mechanism to automatically select a strategy decreases execution time depending on garbage collection overhead (itself dependent on partition size, cache size and strategy), for example, with very limited main memory resource ($\sim$1-2GB) and where garbage collection is frequently triggered, we recorded a decrease of up to 25% in execution time compared to the default mechanism.

MLlib [25] is a set of distributed machine learning libraries written in Scala using the Spark runtime. DistributedWekaSpark can be viewed as a distributed framework that is as powerful as MLlib yet both utilizes existing Weka code and is immediately usable on distributed systems by existing Weka users. An area of further work is to compare performance between DistributedWekaSpark and MLlib.

ACKNOWLEDGMENT

The work was supported by an IBM Faculty Award in Big Data Engineering. The authors wish to thank Dr Mark Hall at the University of Waikato for his advice and encouragement.

REFERENCES

[1] M. Beyer and D. Laney, "The importance of big data: A definition," Stamford, CT:Gartner.

[2] "Apache Hadoop," http://hadoop.apache.org/.

[3] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Com ACM*, pp. 107–113, 2008.

[4] "Apache Spark," https://spark.apache.org/.

[5] S. Sakr, A. Liu, and A. G. Fayoumi, "The Family of Mapreduce and Large-scale Data Processing Systems," *ACM Comput. Surv.*, vol. 46, no. 1, pp. 11:1–11:44, 2013.

[6] "SparkR," http://amplab-extras.github.io/SparkR-pkg/.

[7] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan, "SystemML: Declarative Machine Learning on MapReduce," in *Interntl Conf. on Data Engineering*, 2011, pp. 231–242.

[8] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA Data Mining Software: An Update," *SIGKDD Explor. Newsl.*, pp. 10–18, 2009.

[9] R. A. Muenchen, "The Popularity of Data Analysis Software," http://r4stats.com/articles/popularity//, accessed: 2015-02-02.

[10] D. Wegener, M. Mock, D. Adranale, and S. Wrobel, "Toolkit-Based High-Performance Data Mining of Large Data on MapReduce Clusters," in *ICDM*, 2009, pp. 296–301.

[11] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing," in *NSDI*, 2012.

[12] U. M. Fayyad, G. Piatetsky-Shapiro, and P. Smyth, "From Data Mining to Knowledge Discovery: An Overview," in *Advances in KDDM*, 1996, pp. 1–34.

[13] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques, 3rd Edition*, 2011.

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

[15] R. Agrawal and J. C. Shafer, "Parallel Mining of Association Rules," *Knowl. and Data Eng.*, pp. 962–969, 1996.

[16] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Disk-locality in Datacenter Computing Considered Irrelevant," in *Conf. on Hot Topics in Operating Systems*, 2011.

[17] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron, "Scale-up vs Scale-out for Hadoop: Time to Rethink?" in *Cloud Computing*, 2013, pp. 20:1–20:13.

[18] C. C. Aggarwal, Ed., *Data Classification: Algorithms and Applications*. Chapman and Hall/CRC, 2014.

[19] "Apache Mahout," http://mahout.apache.org/.

[20] E. R. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. E. Gonzalez, M. J. Franklin, M. I. Jordan, and T. Kraska, "MLI: an API for distributed machine learning," *ICDM*, 2013.

[21] *Radoop: Analyzing big data with rapidminer and hadoop*, 2011.

[22] S. Das, Y. Sismanis, K. S. Beyer, R. Gemulla, P. J. Haas, and J. McPherson, "Ricardo: Integrating R and Hadoop," in *Intl Conf. on Management of Data*, 2010, pp. 987–998.

[23] "RHIPE," https://www.datadr.org/, accessed: 2015-03-03.

[24] H. Lin, S. Yang, and S. Midkiff, "RABID: A distributed parallel R for large datasets," in *Congress on Big Data*, 2014, pp. 725–732.

[25] "MLib," https://spark.apache.org/mllib/.

[26] M. Prez, A. Snchez, P. Herrero, V. Robles, and J. Pea, "Adapting the Weka Data Mining Toolkit to a Grid Based Environment," *Web Intelligence*, pp. 492–497, 2005.

[27] S. Celis and D. Musicant, "Weka-parallel: machine learning in parallel," Carleton College, Tech. Rep., 2002.

[28] D. Talia, P. Trunfio, and O. Verta, "Weka4WS: A WSRF-Enabled Weka Toolkit for Distributed Data Mining on Grids," *Knowledge Discovery in Databases*, pp. 309–320, 2005.

[29] A. McCallum, K. Nigam, and L. H. Ungar, "Efficient Clustering of High-dimensional Data Sets with Application to Reference Matching," in *KDD*, 2000, pp. 169–178.