

COMP390-09A Report

Distributed Machine Learning with Hadoop

Nick Jenkin

2009

Supervisor: Bernhard Pfahringer

Abstract

Machine learning datasets are becoming larger, as such methods are required to distribute algorithms across multiple CPU's or cores, but also across clusters of machines. This project explores the use of MapReduce [1] for distributed machine learning, and proposes a technique for the distribution of algorithms within the popular machine learning framework, Weka [2] without modification. The method used shows a significant performance increase over a standalone instance of Weka, as well as an increase in accuracy for some algorithms.

Contents

1	Introduction	5
2	Background	6
2.1	MapReduce	6
2.1.1	Hadoop	7
2.2	Distributed Machine Learning	7
2.3	Hadoop on Symphony: Word Count	8
2.3.1	Experiment	8
2.3.2	Results	8
3	Architecture	11
3.1	Classifier	11
3.2	Evaluator	11
3.3	Invocation	11
4	Algorithms	12
4.1	Naïve Bayes	12
4.2	Generic Distribution with Bagging	12
4.2.1	Stratifying Datasets	13
4.2.2	Weka Bridge	15
4.2.3	Evaluation	15
5	Experimental Results	16
5.1	Weka Performance and Accuracy Comparison	16
5.1.1	Results	16
5.2	Stratification Group Size Effect	16
5.2.1	Results	17
5.3	Weka-Hadoop Dataset Performance	18
5.3.1	Results	18
6	Conclusions and Future Work	20

List of Figures

1	MapReduce distribution technique	6
2	Completion time for varying numbers of nodes and dataset sizes	9
3	Completion time for 32MB and 64MB HDFS block sizes on a 15GB dataset	10
4	Completion time for 32MB and 64MB HDFS block sizes on a 30GB dataset	10
5	Overall system design	11
6	Completion time for varying numbers of nodes and dataset sizes	19

List of Tables

1	Result of stratification when group size is 16	13
2	Comparison of Weka and Hadoop-Weka run times and accuracy .	17

3	Group Size and Accuracy using JRip, alpha dataset. Time is recorded in seconds	17
4	Group Size and Accuracy using J48, alpha dataset. Time is recorded in seconds	18
5	Group Size and Accuracy using Naïve Bayes, alpha dataset. Time is recorded in seconds	18
6	Datasets used for Hadoop/Symphony evaluation	18

List of Algorithms

1	Word Count MapReduce Task	8
2	Naïve Bayes MapReduce algorithm	12
3	Stratification: Step 1: Global Distribution	14
4	Stratification: Step 2: Assign groups	14
5	Stratification: Step 3: Sort file by group id	15

1 Introduction

This project explores distributed machine learning using Hadoop, a MapReduce [1] framework. MapReduce provides a simplistic approach to distributed computing allowing developers to ignore many of the complications a typical distributed application has. While the approach to distribution in MapReduce is certainly simplistic, algorithms still require a lot of thought into how to implement it in a MapReduce style.

Hadoop was used to create a machine learning framework to perform classification and evaluation of distributed classifiers. The framework provided utility functions to the classifiers, such as Hadoop compatible dataset parsers. This allowed classifiers to be written with minimal interaction with Hadoop, but freedom was also given for classifiers which require multiple MapReduce jobs – such as when chaining jobs together.

Due to it being difficult to implement some algorithms in a MapReduce style, a generic distributed method is proposed. This method uses a bagging-like process to split files across a cluster. Taking this method further, it is integrated into the Weka [2] machine learning framework to allow for the distribution of Weka algorithms across a cluster without modification. This distribution method was evaluated to determine how well it scales, with varying numbers of nodes and dataset sizes. As well as how it compares to non-distributed Weka.

2 Background

In this section MapReduce is explained, as well as Hadoop, the implementation of MapReduce used in this report. In addition to this, some previous distributed machine learning projects are discussed. This section also evaluates the University of Waikato cluster's performance with a simple MapReduce application.

2.1 MapReduce

MapReduce is a programming model designed to simplify the concepts around large scale distributed computing [1]. A MapReduce system takes care of the distribution of tasks across a number of machines, while the user is only required to create two functions, *map* and *reduce*.

The *map* function takes a single instance of input, as a key/value pair. The function then outputs key/value pairs which are grouped by key, by the underlying library, and sent to the *reduce* function.

The *reduce* function takes a key, and a list of values which have been collected from the *map* function. Using this list of values it performs some computation over that list, and outputs key/value pairs based on that computation. There is no restriction on the number of key/value pairs being output.

MapReduce is suited to dealing with large datasets ranging from gigabytes to petabytes. This makes it ideal for data mining datasets which do not fit into physical memory. The major pitfall to using MapReduce is that it is suited to applications or algorithms that do additive analysis. This can be overcome by chaining multiple MapReduce tasks together, but in some situations it would be more efficient to use an alternate distribution method.

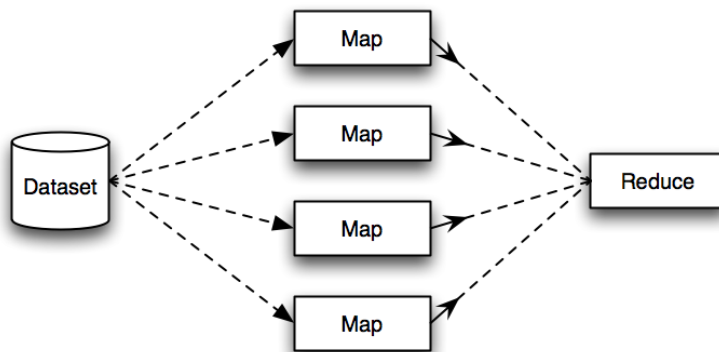


Figure 1: MapReduce distribution technique

Figure 1 shows how MapReduce distributes an application. First the input dataset is split into n splits, where n could be determined by the size of the dataset and the number of nodes. The splits are then distributed to each compute node. This is typically done implicitly by the use of a distributed file system, with compute nodes being optimally allocated blocks of data which they already have. Each compute node starts executing the *map* function over the subset of data and produces a set of intermediate key/value pairs, which will be

sent to the *reduce* nodes. When using multiple *reduce* nodes, the output is split across multiple files. Section 2.3 demonstrates a simple MapReduce application.

2.1.1 Hadoop

Hadoop [3] is an open-source Java implementation of MapReduce [1]. This project uses Hadoop as the MapReduce system, but there are several other implementations such as Disco [4]. Hadoop was chosen because it is currently the most feature-complete system and is widely used in industry.

The Hadoop framework comes with two components, the MapReduce functionality and a distributed file system (HDFS) [5]. The distributed file system is used to distribute input data across the cluster. Hadoop tries to allocate map tasks based on the physical location of each piece of data on HDFS. Hadoop also has the ability to be rack-aware, this allows further optimisation and reduction in network traffic by allocating map tasks based on network proximity between blocks of data.

A typical installation of Hadoop on a cluster has two master nodes, one for MapReduce and one for HDFS.

2.2 Distributed Machine Learning

Distributed machine learning with MapReduce is not new, there are several existing projects, such as Apache Mahout [6]. Apache Mahout is still in the early stages of development, and has implemented only algorithms described by Chu et al. [7]. Apache Mahout is certainly a project to watch, as it is currently one of the only machine learning projects dedicated to creating a truly distributed machine learning framework.

Chu et al. [7] published ten MapReduce implementations of machine learning algorithms: Locally Weighted Linear Regression, Naive Bayes, Gaussian Discriminative Analysis, k-means, Logistic Regression, Neural Network, Principal Components Analysis, Independent Component Analysis, Expectation Maximization and Support Vector Machines. The work primarily focused on using MapReduce as a method of distribution on a single, multi-core machine – but there is no reason they cannot be applied to a cluster of machines.

MapReduce has also been used to integrate distributed machine learning algorithms into Weka [8, 2]. This project involved adding support for MapReduce based algorithms into Weka, but it also implemented several algorithms described by [7].

MapReduce is certainly not the only method in which distributed machine learning takes place, there are several add-ons to Weka [2], such as Weka4WS [9] that allow for distribution. Weka4WS allows distribution of three common tasks in Weka: Labelling, where class labels are assigned to instances based on a predicted model; Testing, where an algorithm's prediction accuracy is calculated by labelling a known set of instances; and cross validation, a testing method which splits an input dataset up into n partitions. These three methods are all ideal for simple distribution as partitioning the datasets used across multiple machines will not change the end result.

2.3 Hadoop on Symphony: Word Count

The University of Waikato has a cluster, called Symphony, which consists of ninety nodes. Each node runs 64-bit Linux and has a dual core 2.66GHz CPU, 4GB of RAM, and a local scratch disk. Because Symphony is relatively new, it was of interest to determine how well Hadoop performed on Symphony. To do this a word count task was tested, which simply processes a document and returns the number of times each word in the document appears. This parallels with data mining as many data mining tasks are simply processing a dataset and collating results.

Algorithm 1 Word Count MapReduce Task

```
function map(key, line, output):  
  for all words in line do  
    output.collect(word, 1)  
  end for  
  
function reduce(word, values, output):  
  count = SUM(values)  
  output.collect(word, count)
```

Algorithm 1 shows the *map* and *reduce* functions for the word count task. The *map* function takes a line of text, and for every word in that line it will output a key/value pair where the key is the word and the value is one. The *reduce* function is given a word, and a list of values (for this task every value will be one), it sums the values and outputs the word count for the word provided.

2.3.1 Experiment

The experiment tested two variables, the number of nodes used and the size of the dataset. The number of nodes configured were: 10, 20, 30, 40, and 50. The dataset sizes were: 15GB, 30GB, and 60GB. The performance of the varying configuration was determined by the time it took to complete a given configuration. An additional experiment was performed to determine the affect block size had on performance. Block size is used by the distributed file system to split a single file into blocks. The default value is 64 megabytes, we reduced this to 32 megabytes and used the 30GB dataset, with 10, 20, 30, 40, and 50 nodes.

2.3.2 Results

Figure 2 shows the results for the different combinations of nodes and dataset sizes. The results indicate there is a significant performance gain in using up to 30 nodes. But the gains of using 30 or more nodes were minimal for this experiment. The cause of this reduction in performance gain can be explained by the limited number of reducer nodes used in this experiment. This ultimately means the reducer nodes become the bottlenecks. Ultimately the number of reducers used is algorithm specific, for algorithms which output a large number of key/value pairs, it may be appropriate to increase the number of reducer nodes.

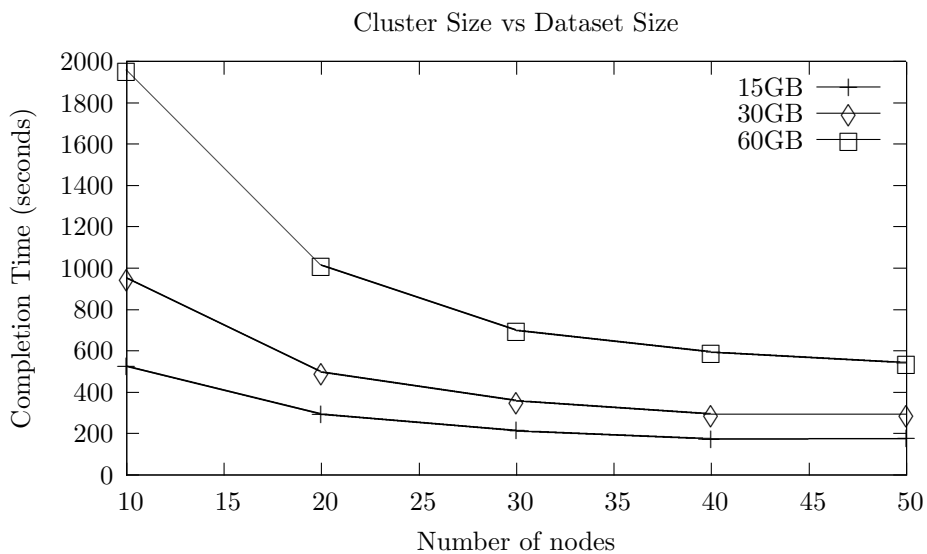


Figure 2: Completion time for varying numbers of nodes and dataset sizes

Figure 3 and Figure 4 show the effect the distributed file system’s block size has on performance. Two block sizes were tested: 32MB and 64MB (the hadoop default). The results indicate there is a minor performance gain in having a larger block size. Ultimately smaller block sizes will increase the likelihood of network activity. This is due to how *map* tasks are allocated to compute nodes. Because Hadoop tries to allocate based on the physical location of a block in the distributed file system, an increased number of blocks results in a higher block to compute node ratio, resulting in increased competition for compute nodes.

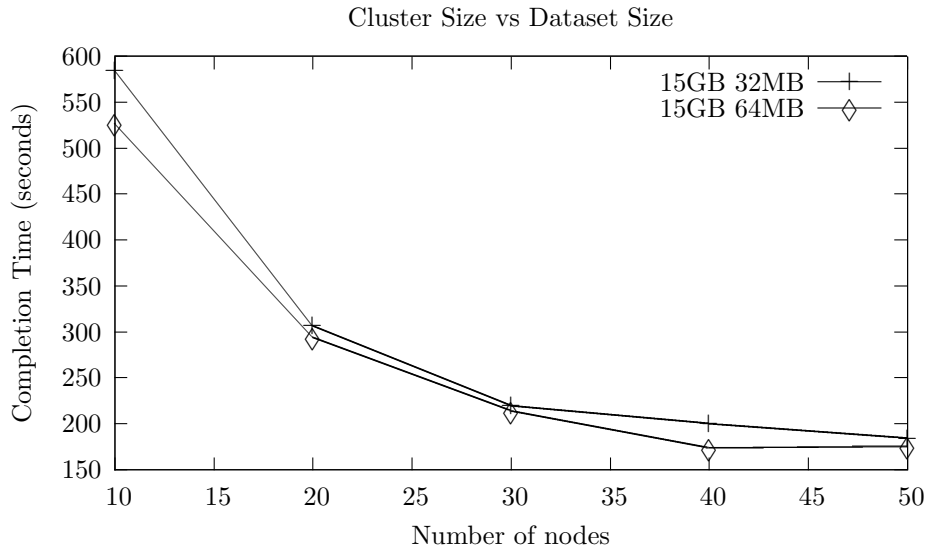


Figure 3: Completion time for 32MB and 64MB HDFS block sizes on a 15GB dataset

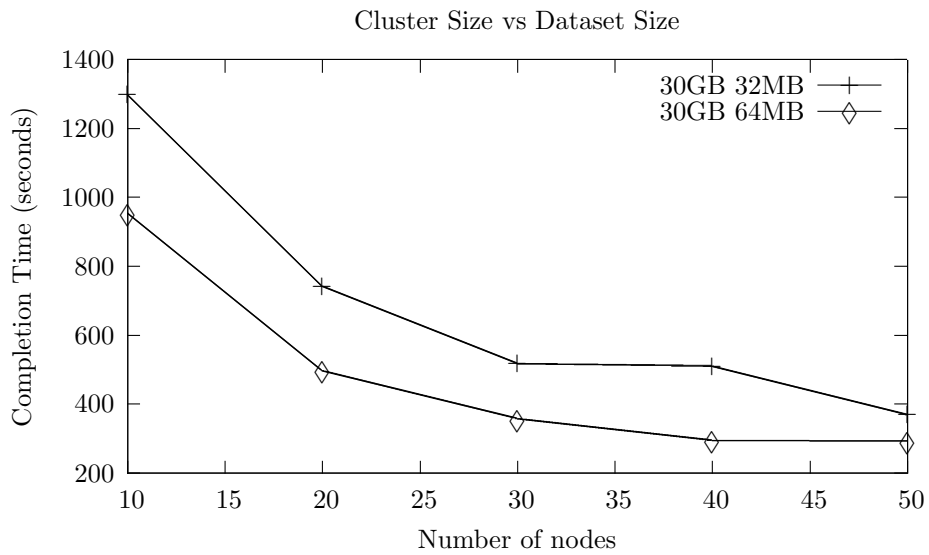


Figure 4: Completion time for 32MB and 64MB HDFS block sizes on a 30GB dataset

3 Architecture

The main goal of the underlying framework was to provide a method of distributing machine learning tasks and analysing results with Hadoop. To do this the system was separated into two parts, the Classifier and the Evaluator. The two parts were connected by a Model, which is defined when creating a Classifier. Datasets were formatted as CSV files, but a parser for Hadoop was created to extract individual rows into an instance object. This meant classifiers did not have to parse any datasets, as each *map* task receives an instance object.

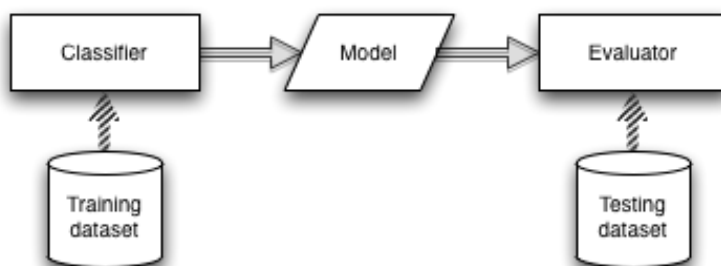


Figure 5: Overall system design

Figure 5 shows the general process in which the system operates, with the model connecting the classification and evaluation steps.

3.1 Classifier

The classifier is a user-defined algorithm which performs some processing on the given input to produce a model. Each classifier has the ability to start multiple MapReduce tasks, this allows for MapReduce tasks to be joined together, which is required for many algorithms. When creating a classifier it is required that a model be defined, which is later used in evaluation. The model is initialised by processing the output from the final MapReduce task the classifier executes.

3.2 Evaluator

The evaluator calculates several statistics on the accuracy of a given classifier. Evaluation is also performed using a MapReduce cluster, this allows for the use of large testing datasets. The evaluator distributes the result from the classifier (model) to each node in the cluster. Each node initialises a model which is then used for classification.

3.3 Invocation

The system is invoked by using the *hadoop* command provided with the Hadoop framework. The command allows you to execute a given jar file, as well as set various Hadoop settings. The machine learning framework takes as arguments the input, output, and testing datasets as well as what classifier to use.

4 Algorithms

There are several algorithms which have been successfully implemented on MapReduce, these include: Locally Weighted Linear Regression, Naïve Bayes, Gaussian Discriminative Analysis, k-means, Logistic Regression, Neural Network, Principal Components Analysis, Independent Component Analysis, Expectation Maximization and Support Vector Machines. This project however, will only look into Naïve Bayes as a pure MapReduce implementation. A generic distribution technique is also described, which allows for the distribution of algorithms not written for MapReduce.

4.1 Naïve Bayes

Naïve Bayes (NB) is a simple algorithm that fits well with the additive style MapReduce requires. Because NB only requires means and standard deviations, these can be calculated in a single MapReduce iteration over the dataset. Algorithm 2 shows a pseudo-code implementation of the *map* and *reduce* functions. In order to calculate the means and standard deviations the global class distribution needs to be known. In addition to this attribute-class values need to be grouped, and provided to the *reduce* function.

Algorithm 2 Naïve Bayes MapReduce algorithm

```
function map(key, instance, output):
  i = 0
  for all attributes in instance do
    output.collect(CONCAT(instance.target, i), attribute)
    i++
  end for
  output.collect(CONCAT("Target", instance.target), 1)           ▷ global dist.

function reduce(key, values, output):
  if key is target key then
    output.collect(key, SUM(values))
  else
    sum = SUM(values)
    sumSq = SUM(square of every value in values)
    count = number of values
    output.collect(CONCAT(key, "mean"), sum / count)             ▷ mean
    stddev = sqrt(abs(sumSq - mean * sum) / count)
    output.collect(CONCAT(key, "stddev"), stddev)               ▷ standard deviation
  end if
```

4.2 Generic Distribution with Bagging

While investigating algorithms to implement in MapReduce it became obvious that there were many algorithms that were not suited to the MapReduce model. MapReduce tended to favour additive algorithms, that is, algorithms that could look at each instance independently and produce some metric for that instance which could later be added to the global model. It also required a great deal

of thought into how one would implement an algorithm in a MapReduce style. So to allow for a way to distribute machine learning algorithms without the need to rewrite them into a MapReduce format, a classifier was created to use a bagging-like process to split a large dataset up into many smaller ones, which could then use voting for classification. The obvious downside to this technique is that the entire dataset will not be processed by a single classifier, but because bagging is an accepted technique in machine learning it may be an acceptable compromise.

4.2.1 Stratifying Datasets

In order to perform bagging, input instances needed to be allocated into groups. Due to limitations in how input data is received by each *map* task, traditional replacement bagging was not possible. As such a MapReduce task was created to allocate instances into groups, and these groups were representative of the global class distribution. This ensured that each classifier would receive a similar distribution of classes as if it were processing the entire dataset.

The stratification process will ensure that at least one class type is in each group, and to do this it will duplicate instances when required. There are several reasons to do this, firstly it makes evaluation easier, as we can guarantee that each classifier has seen every possible class. But also if an attribute has minimal representation in the dataset, it may be because it is an extreme outlier, allowing it to be easily identified.

Class	a	b	c	d
Global	100	50	10	1
Subset 1	10	5	1	1
...				
Subset 10	10	5	1	1

Table 1: Result of stratification when group size is 16

Figure 1 shows the result of the stratification process when the group size is 16. The size of the groups can be specified at run time, and the optimal values can vary from dataset to dataset. This is because the groups are loaded entirely into memory for processing, as such the groups must be able to fit in memory.

The process of stratifying the datasets was performed using MapReduce. The task consisted of three MapReduce operations: obtain the global distribution, assign instances groups, and finally sort the instances by group. The output of the stratifying process is a CSV file of instances, with a group prefix for each instance. A file parser was created for Hadoop to process this format into groups of instances.

Algorithm 3 shows the *map* and *reduce* operations for obtaining the global distribution of a dataset. This is a simple task, whereby the number of instances per class is counted. The result from this task is used in Algorithm 4 to assign groups to each instance.

Algorithm 4 shows the *map* and *reduce* operations for assigning groups to instances. The *map* operation in this step simply groups each instance by class.

Algorithm 3 Stratification: Step 1: Global Distribution

```
function map(key, instance, output):  
    output.collect(instance.target, 1)  
  
function reduce(key, values, output):  
    count = SUM(values)  
    output.collect(key, count)
```

The *reduce* operation takes the output from Algorithm 3 and uses this to determine how many instances of each target belong in a group. In situations where there are not enough instances for a particular class, they will be duplicated to ensure there is at least one class type per group. The β , α , and γ values are obtained from the output of Algorithm 3. While the δ value is a user defined parameter to indicate the group size.

Algorithm 4 Stratification: Step 2: Assign groups

```
function map(key, instance, output):  
    output.collect(instance.target, instance)  
  
 $\beta$  = total instances / group size; ▷ number of groups  
 $\alpha$  = distribution of class  
 $\gamma$  = number of instances per class  
 $\delta$  = group size  
function reduce(key, values, output):  
    required =  $\delta \times \alpha \times \beta$ , duplication = 1, extras = 0, gid = 0  
if required >  $\gamma$  then  
    duplication = required /  $\gamma$   
    if duplication - FLOOR(duplication) > 0 then  
        extras =  $\gamma \times$  (duplication - FLOOR(duplication))  
        extraPerInstance = extras /  $\gamma$   
    end if  
end if  
for instance in values do  
    for j=0; j < duplication + extraPerInstance; j++ do  
        output.collect(gid, instance)  
        gid++  
        if gid >  $\beta$  then  
            gid = 0  
        end if  
    end for  
end for
```

Algorithm 5 sorts the output from Algorithm 4 by group id. This is required for the parser used in classification as it expects the input file to be sorted by group.

Algorithm 5 Stratification: Step 3: Sort file by group id

```
function map(key, instance, output):  
    output.collect(instance.group_id, instance)  
  
function reduce(key, values, output):  
for instance in value do  
    output.collect(key, instance)  
end for
```

4.2.2 Weka Bridge

The generic distribution allows for classifiers not written specifically for MapReduce to be distributed. This allows for a bridge between Hadoop and Weka [2] to be created to allow for the distribution of classifiers built into Weka without modification. Weka is typically memory-bound, but also many algorithms in Weka do not scale well with large datasets. By distributing the classifiers over many nodes, it allows for the dataset each classifier processes to be reduced, thus reducing memory usage and improving the performance of some algorithms.

The Weka bridge provided the appropriate data to Weka classifiers to be trained. In order to evaluate the Weka classifiers, they were serialized using with Java's serialization API and packaged into a model, which could then use the standard evaluation method described in Section 3.2.

4.2.3 Evaluation

Evaluation is performed using voting, a classifier is built for every group, and the classification of an instance is obtained by taking the majority predicted class. It should be noted that during testing, in many cases the evaluation step took longer to complete than the building of the classifiers. This is because if there are for example, 200 groups, the instance is classified 200 times and the result is then determined. As such, when deciding on the group size (Section 4.2.2) evaluation time should also be considered.

5 Experimental Results

Three experiments were performed on the Weka-Hadoop bridge (Section 4.2.2) to determine the performance gain over traditional standalone Weka [2]; the effect the group size (Section 4.2.1) has on accuracy and performance; and the Weka-Hadoop performance across varying numbers of nodes, and varying dataset sizes. These experiments were performed on the University of Waikato cluster (Section 2.3). This is a shared cluster, and as such the results below can vary depending on the amount of people using the cluster at the same time the experiments were run.

One aspect of MapReduce which is not covered in these experiments is the effect the number of reducer nodes has on performance. In all the experiments only one reducer node was used. Increasing the number of reducer nodes used can benefit algorithms which output a large number of key/value pairs.

5.1 Weka Performance and Accuracy Comparison

This experiment aims to determine the difference in performance and accuracy between classification using Weka [2] and the Hadoop-Weka (Section 4.2.2) implementation. The experiment was performed using 30 nodes of the Symphony cluster for Hadoop-Weka, as well as a Dual Xeon 3GHZ, 6GB machine for the standalone Weka classifier. Due to the limited amount of memory on the standalone machine, only a single dataset was able to be tested: alpha, which contains 250,000 instances with 500 numeric attributes each. The stratification group size (Section 4.2.1) was 10,000 for the Hadoop-Weka dataset. Three algorithms were tested: JRip (RIPPER [10]), J48 (C4.5 [11]) and Naïve Bayes. Although due to JRip being a quadratic algorithm, data was unable to be collected for the Weka-JRip run. The default Weka settings were used for all the algorithms. The time taken to train and evaluate the datasets was recorded, as well as the accuracy.

5.1.1 Results

Table 2 lists the results from the experiment. For all the known values, the Weka-Hadoop bridge provided a higher accuracy, as well as a significant improvement in speed. Distributed J48 was completed in $\frac{1}{100}$ the time of the non-distributed experiment, while also increasing the accuracy by 2.83%. The only algorithm which did not show a large improvement in performance and accuracy is Naïve Bayes (NB). Due to the simplicity of the NB algorithm, it only iterates over the dataset once, where both JRip and J48 can iterate of the dataset many times.

5.2 Stratification Group Size Effect

The purpose of this experiment is to determine what effect the group size used in the stratification process (Section 4.2.1) has on the performance and accuracy of three Weka [2] algorithms: JRip (RIPPER [10]), J48 (C4.5 [11]) and Naïve Bayes. The default Weka settings were used for all the algorithms.

The experiment evaluated the alpha dataset, which contains 250,000 training examples and 250,000 test examples, with 500 attributes each. The group

System	Algorithm	Time (Seconds)	Accuracy
Weka	JRip	>> 3 days	unknown
Weka	J48	29881	55.44%
Weka	Naïve Bayes	724	54.87%
Hadoop	JRip	422	60.15%
Hadoop	J48	354	58.27%
Hadoop	Naïve Bayes	230	55.93%

Table 2: Comparison of Weka and Hadoop-Weka run times and accuracy

sizes tested were: 1000, 2000, 5000, 10,000, and 15,000. The experiments were repeated two times and the timing results were averaged. The Hadoop cluster was setup to contain 30 nodes, with each node having two workers.

5.2.1 Results

Table 3 shows the results from the JRip algorithm. The results indicate that for smaller group sizes, training time is faster. This is because there is reduced processing required by the classifier to determine a model. For larger group size values, there is a lot more work required to determine an outcome, and this is compounded by the fact that JRip is $O(n^2)$. The evaluation times however are larger for smaller group sizes. This is because smaller group sizes produce more classifiers, for example on the alpha dataset, when the group size is 1000, there are about 250 classifiers, while at 15000 there are 17.

The accuracy of JRip was best with a group size of 2000, although it was a minor improvement over 5000. A reduction in accuracy can be seen for increasing group size values. The group size of 1000 shows a decrease in accuracy, this indicates that 1000 examples may not be enough to create an ideal model for this classifier.

Group Size	Training Time	Test Time	Total Time	Accuracy
1000	68s	3371s	3439s	60.89%
2000	87s	1617s	1704s	61.10%
5000	130s	506s	636s	61.09%
10000	271s	124s	395s	60.15%
15000	520s	54s	574s	56.22%

Table 3: Group Size and Accuracy using JRip, alpha dataset. Time is recorded in seconds

Table 4 shows the results for the J48 classifier. The results for training and evaluation are similar to that of JRip (Table 3). The primary difference is with the accuracy, as unlike JRip, J48 has a clear correlation between group size and accuracy. In all cases, the smaller the group size, the better the accuracy was.

Table 5 shows the results for the Naïve Bayes classifier. This algorithm is faster than both JRip (Table 3) and J48 (Table 4), this is because the algorithm is linear ($O(n)$). The accuracy however shows very little improvement for the varying group sizes, with all the results coming within a 0.44% range.

Group Size	Training Time	Test Time	Total Time	Accuracy
1000	69s	3396s	3465s	65.15%
2000	85s	1600s	1685s	64.93%
5000	127s	477s	604s	62.66%
10000	162s	125s	287s	58.27%
15000	336s	54s	381s	54.04%

Table 4: Group Size and Accuracy using J48, alpha dataset. Time is recorded in seconds

Group Size	Training Time	Test Time	Total Time	Accuracy
1000	32s	4257s	4289s	54.93%
2000	30s	1955s	1985s	54.86%
5000	49s	615s	664s	54.85%
10000	38s	144s	182s	54.93%
15000	232s	60s	292s	54.49%

Table 5: Group Size and Accuracy using Naïve Bayes, alpha dataset. Time is recorded in seconds

5.3 Weka-Hadoop Dataset Performance

This experiment aims to see how the Weka-Hadoop approach (Section 4.2.2) scales with larger datasets and varying numbers of nodes. Table 6 lists the three datasets used in this experiment, with alpha being the smallest, and dna being the largest. The experiment collected the training time of J48 (C4.5 [11]) over clusters of size: 10, 20, 30, 40, and 50. The default J48 Weka settings were used. Evaluation time was not calculated because the group size used was 10,000. While this group size is appropriate for the alpha and zeta datasets, it results in 2500 classifiers being created for the dna dataset. This number of classifiers would take a significant amount of time to complete evaluation. Larger group sizes were not tested due to memory constraints on the cluster nodes.

Name	Size	Instances	Attributes	Classes
alpha	1.2GB	250,000	500	2
zeta	4.5GB	250,000	2000	2
dna	9.4GB	25,000,000	200	2

Table 6: Datasets used for Hadoop/Symphony evaluation

5.3.1 Results

Figure 6 shows the results from this experiment. The alpha and zeta datasets did not show any considerable gain from additional nodes in the cluster. This suggests that a ten node cluster would be sufficient for computation of datasets

up to 5GB in size. The dna dataset however showed a significant improvement in performance until the cluster size reached 40. There was no significant gain between 40 and 50 nodes. The reasons for this could be that the reduce stage became the bottleneck, if this is the case increasing the number of reducers would improve performance.

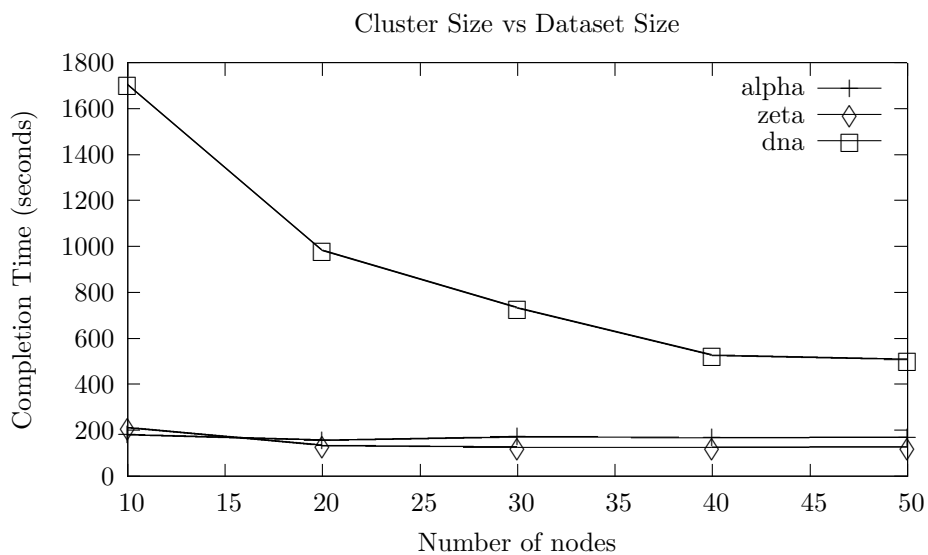


Figure 6: Completion time for varying numbers of nodes and dataset sizes

6 Conclusions and Future Work

This project has explored the use of MapReduce for distributed machine learning. The Weka-Hadoop bridge (Section 4.2.2) provides a simple method for the distribution of Weka algorithms, without modification over a cluster of machines using Hadoop. Overall the results from the experiments indicate a significant increase in performance, but also an increase in accuracy for some algorithms. The Weka-Hadoop bridge allows users of Weka to perform machine learning tasks, which would not normally be viable on a single CPU machine, or a machine with limited memory.

There are many benefits to the generic distribution approach, as it does not require redevelopment of algorithms with distribution in mind. As such it allows people with limited distributed computing experience to perform large machine learning computations, without significant effort. But it also allows for algorithms to be distributed which would not otherwise work with MapReduce, or some other distribution system.

This project has also evaluated the performance of MapReduce, for both simple data analysis tasks and machine learning tasks on the University of Waikato cluster. Overall the results from the experiments on the cluster indicate that it is an ideal environment for MapReduce tasks.

Areas which could be explored further are ways to reduce the effect the group size (Section 4.2.1) has on evaluation times. Methods to do this may include discarding poor performing classifiers, or prematurely determining a classification if the vote is obviously heading for a specific class. The impact of reduce nodes was also not covered in this report, and this is an area where performance gains might be easily achieved by just increasing the number of reduce nodes.

Overall the Hadoop appears to be a suitable solution for distributed machine learning. It provides a simplistic layer upon which distributed applications can be developed without the need to worry about synchronisation, redundancy or many of the other obstacles in distributed computing. The main problem is designing an algorithm in a MapReduce style, but if that doesn't work out – the generic distribution approach works well.

References

- [1] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters.
- [2] I.H. Witten and E. Frank. Data mining: practical machine learning tools and techniques with Java implementations. *ACM SIGMOD Record*, 31(1):76–77, 2002.
- [3] Apache Software Foundation. Apache Hadoop. <http://hadoop.apache.org>.
- [4] S. Papadimitriou and J. Sun. DisCo: Distributed Co-clustering with Map-Reduce.
- [5] D. Borthakur. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 2007.
- [6] Apache Software Foundation. Apache Mahout. <http://hadoop.apache.org>.
- [7] C.T. Chu, S.K. Kim, Y.A. Lin, Y.Y. Yu, G. Bradski, A.Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *Advances in Neural Information Processing Systems: Proceedings of the 2006 Conference*, page 281. MIT Press, 2007.
- [8] Deyaa Adranale. Parallelization of weka the data mining toolkit using hadoop. Master’s thesis, Otto-von-Guericke University of Magdeburg, December 2008.
- [9] D. Talia, P. Trunfio, and O. Verta. Weka4ws: a wsrf-enabled weka toolkit for distributed data mining on grids.
- [10] W.W. Cohen. Fast effective rule induction. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 115–123. Citeseer, 1995.
- [11] J.R. Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann, 1993.

Distributed Machine Learning with MapReduce

(on the University cluster)

Overview

- MapReduce
 - Example
 - Symphony performance
- MapReduce + ML
- MapReduce + Weka

MapReduce

- Two functions, map and reduce
- the map function is applied to every instance of data
- The output of the map function consists of a set of key/value pairs
- Output with the same key is sent to the same reduce call

Hadoop

- Java implementation of MapReduce
- Includes a distributed file system
- Also includes a lot of the sugar functionality in MapReduce, such as rack-awareness and redundancy

Example: Word Count

Map

Key	Value
the	1
a	1
and	1
and	1
the	1
weka	1

Reduce

Key	Values
the	(1,1)
a	(1)
and	(1,1)
weka	(1)

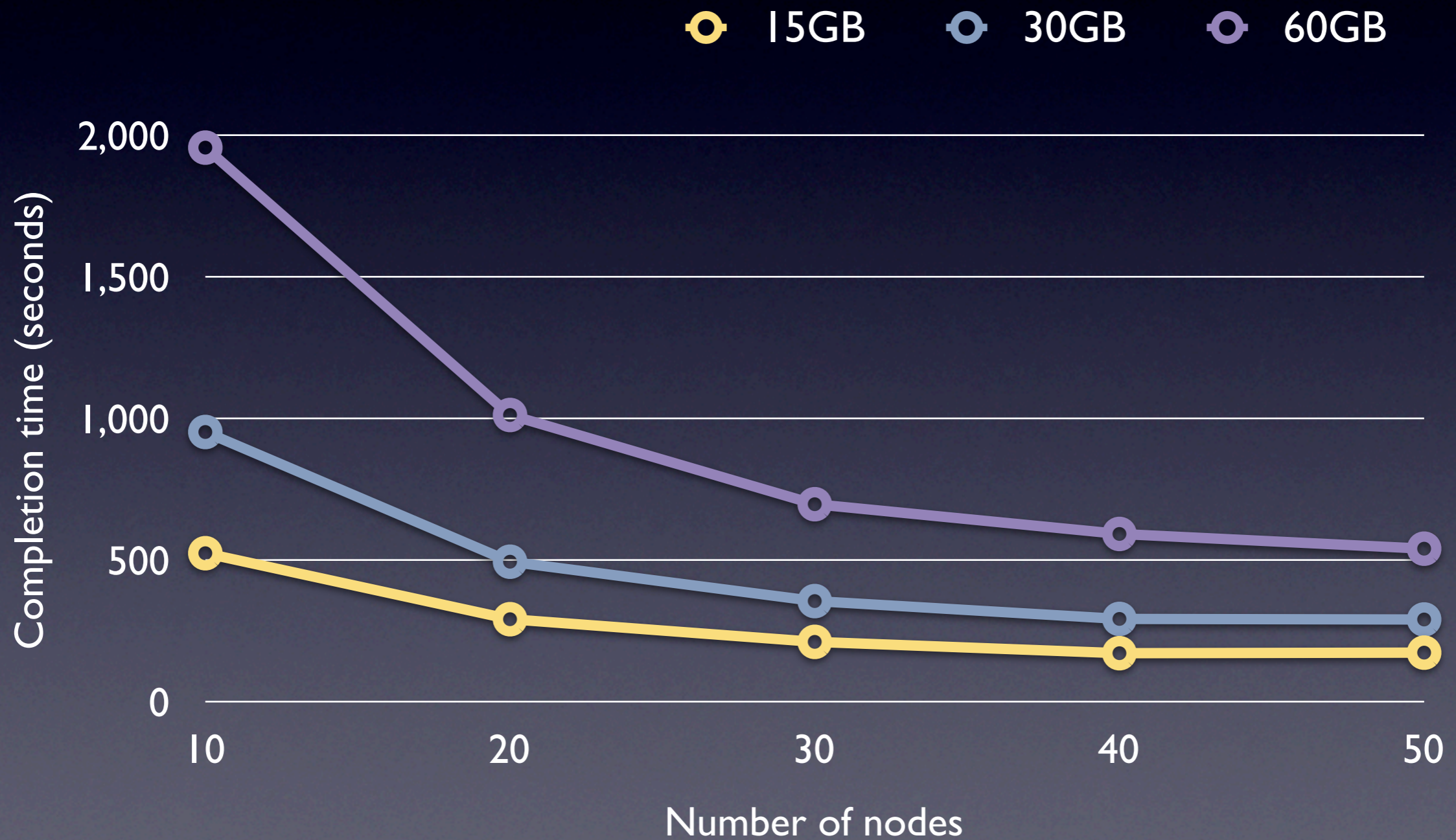
Output

Key	Value
the	2
a	1
and	2
weka	1

Word Count (Code)

```
map(key, line):  
    words = line.split(" ")  
    for word in words:  
        collect(word, 1)  
  
reduce(word, values):  
    collect(word, sum(values))
```

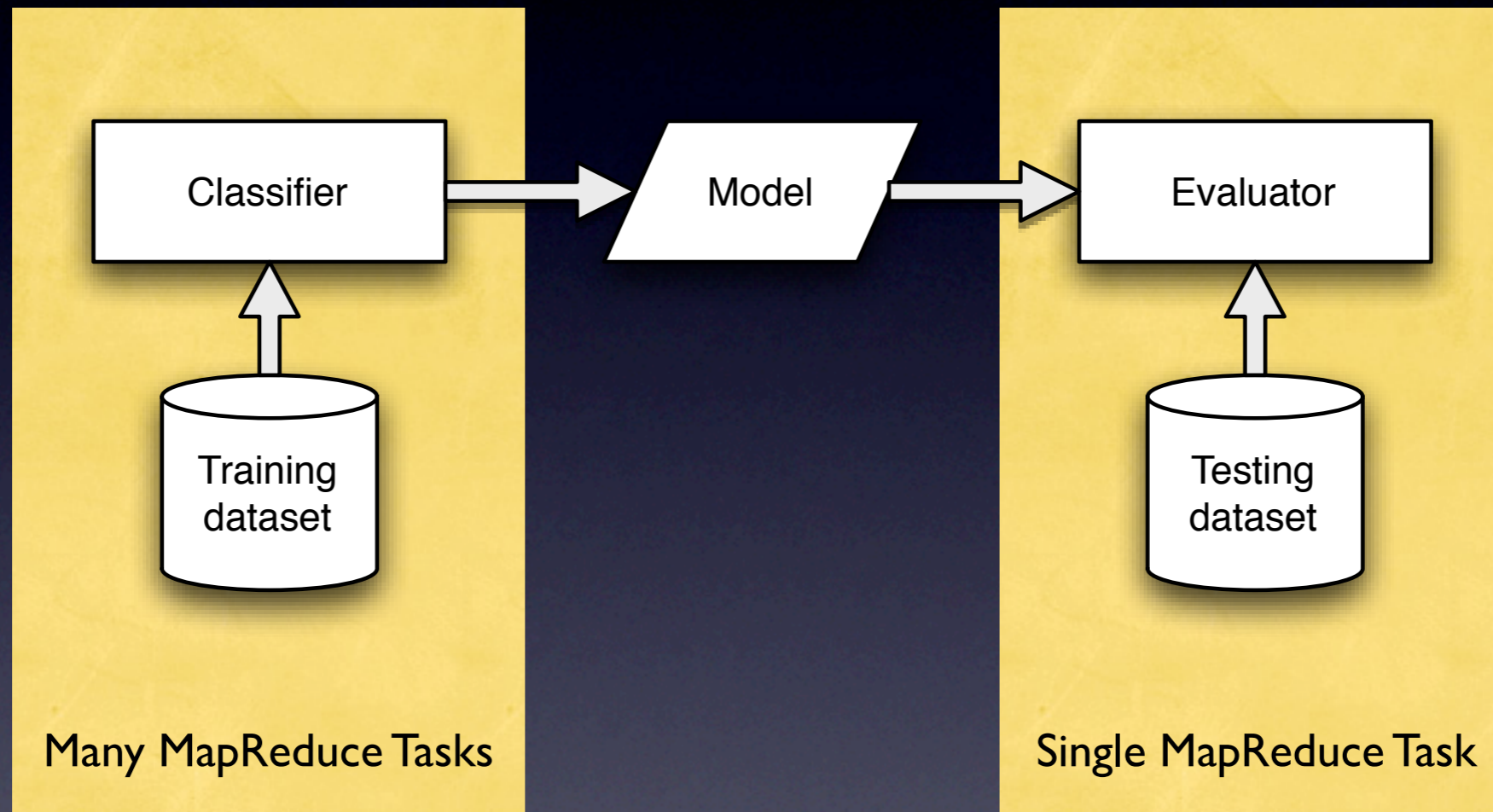
Word Count on Symphony



Machine Learning

- Apache Mahout
 - New project for machine learning with Hadoop
 - Implements Naive Bayes (as well as several clustering algorithms)
- MapReduce is suited to additive algorithms that can process individual instances without knowledge of global state

Architecture



Architecture

- Classifier produces a model based on the input dataset
- Model is distributed to each node
- Evaluator uses model & MapReduce to calculate accuracy statistics.

Naive Bayes

- Additive algorithm, ideal for MapReduce
- Need to compute means and standard deviations
- Can be performed in a single MapReduce operation

Naive Bayes (Code)

```
map(key, instance):
    collect(instance.target, 1)
    for attr in instance.attributes:
        collect(attr.name_instance.target, attr.value)

reduce(key, values):
    if target key:
        collect(key, sum(values))
    else:
        sum = sum(values)
        sumSq = sum(square of every value)
        collect(key_mean, sum/count)
        stddev = sqrt(abs(sumSq - mean * sum)/count)
        collect(key_stddev, stddev)
```

Generalising ML on Hadoop

- Use a bagging-like classifier to distribute algorithms not designed for MapReduce
- Split dataset into subsets, each subset is executed on a single *map* task.
- Use voting to determine classification

Stratifying Datasets

- Instances need to be placed into groups
- A group should have the same/similar class distribution as the global dataset
- Use MapReduce:
 - Obtain global distribution
 - Group by target
 - Assign groups based on target

Weka Bridge

- Distribute Weka classifiers without modification
- Use the bagging-like process to build classifiers
- Serializes the built Weka classes distributes to each evaluation node
- Voting used to determine class

Experiment

- Dataset: alpha, ~250k instances, 501 attributes, two classes
- Weka: Dual Xeon 3Ghz/6GB RAM
- Hadoop: 30 nodes, (each node: 2x2.66Ghz/4GB RAM), 2 workers per node
- Dataset group size for Hadoop was 10,000

Results

Method	Time	Accuracy
Weka JRip	3 days+	???
Hadoop JRip	7 minutes	60.15%
Weka J48	8 hours	55.44%
Hadoop J48	6 minutes	58.27%
Weka Naive Bayes	12 minutes	54.87%
Hadoop Naive Bayes	3 minutes, 50 sec	54.93%

Group Size & Accuracy

JRip

Size	Train	Test	Total	Accuracy
2000	87	1617	1704	61.10%
5000	130	506	636	61.09%
10000	271	124	395	60.15%
15000	520	54	574	56.22%

Group Size & Accuracy

J48

Size	Train	Test	Total	Accuracy
2000	85	1600	1685	64.93%
5000	127	477	604	62.66%
10000	162	125	287	58.27%
15000	336	54	381	54.04%

The End