Smart-MLlib: A High-Performance Machine-Learning Library

Thesis

Presented in Partial Fulfillment of the Requirements for the Degree
Master of Science in the Graduate School of The Ohio State
University

By

David Siegal, B.S.
Graduate Program in Computer Science and Engineering

The Ohio State University

2016

Master’s Examination Committee:

Dr. Gagan Agrawal, Advisor
Dr. Christopher Stewart
Abstract

As the popularity of big data analytics has continued to grow, so has the need for accessible and scalable machine-learning implementations. In recent years, Apache Spark’s machine-learning library, MLlib, has been used to fulfill this need. Spark is a general-purpose cluster computing system with built-in support for iterative applications. Its machine-learning library, which comes packaged with Spark, provides dozens of out-of-the-box, machine-learning implementations on top of the distributed system. While there are clearly benefits of a library like Spark’s MLlib, what is unclear is whether Spark is the best underlying system to use for this type of library.

In this thesis, we present a machine-learning library prototype that is built on top of our own in-Situ MApReduce liTe (Smart) system, Smart-MLlib. Smart has previously been shown to outperform Spark in machine-learning tasks and can offer advantages to Spark in both performance and scalability. The interface developed for Smart-MLlib closely mirrors Spark’s MLlib application programming interface (API), but uses Smart rather than Spark as the underlying system. To test our library’s performance, we built out four machine-learning applications that are also provided in Spark’s MLlib: k-means clustering, linear regression, Gaussian mixture models, and support vector machines. Each of these implementations was tested for efficiency against the version provided in Spark’s MLlib using a variety of input sizes and cluster sizes.
For all four of the applications tested, the speed of Smart-MLlib’s implementation outperformed Spark’s MLlib in every configuration by at least 90%. On average, we outperformed Spark’s MLlib by over 800%. In addition to outperforming Spark’s MLlib in head-to-head comparisons, our library also scaled better than Spark’s MLlib in almost every situation. Specifically, the Smart-MLlib implementations scaled from 4 nodes to 32 nodes an average of 90% to 220% better than Spark’s MLlib for every algorithm tested. These findings show that our machine-learning library enables higher performance than Spark’s MLlib without sacrificing the easy-to-use API.
I would like to start by thanking my advisor, Dr. Gagan Agrawal, for his support and guidance throughout my career at Ohio State. I have been working with Dr. Agrawal since the beginning of my junior year and could not have hoped for a better experience. He masterfully guided me through my research – providing direction when needed, while consistently giving me freedom to explore my own ideas.

Additionally, I would like to thank my family. My parents, Adam and Lorna, for their support and advice throughout my college career. My siblings, Ian and Kimberly, for being such a phenomenal support system. And finally, my wife, Sara, for the countless ways in which she enhances each day.

Lastly, I want to thank the people who helped directly with the production of this thesis: Jia Guo, Kimberly Siegal, and Daniel Marchese. Jia implemented a large part of the SVM-related code for my experiments, Kim edited for grammar, and Dan edited for content. While I am thanking Dan for editing this thesis, I would also like to thank him for his help throughout the last four years. It was incredibly fun having a great friend in many of my classes each semester.
Vita

Summer, 2013 .............................. Infrastructure Project Manager Intern, 
                      Hyland Software
Summer, 2014 .............................. Software Development Engineer Intern, 
                      Audible Inc.
Summer, 2015 .............................. Software Engineer Intern, Facebook 
                      Inc.
May, 2015 ................................. B.S. Computer Science and Engineer- 
                      ing, The Ohio State University
August, 2015 - present ..................... Graduate Research Associate, The 
                      Ohio State University

Fields of Study

Major Field: Computer Science and Engineering
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>iv</td>
</tr>
<tr>
<td>Vita</td>
<td>v</td>
</tr>
<tr>
<td>List of Tables</td>
<td>viii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>ix</td>
</tr>
<tr>
<td>List of Listings</td>
<td>xi</td>
</tr>
</tbody>
</table>

Chapters:

1. Introduction                 | 1    |
   1.1 Motivation                | 2    |
   1.2 Our Contributions        | 2    |
   1.3 Organization             | 3    |

2. Background                  | 5    |
   2.1 MapReduce                | 5    |
   2.2 Spark                    | 7    |
   2.3 Smart                    | 9    |
   2.4 MLlib                    | 12   |

3. Architecture                | 14   |
   3.1 System Overview         | 14   |
   3.1.1 System Advantages     | 17   |
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Smart’s main API functions</td>
<td>11</td>
</tr>
<tr>
<td>4.1 Smart-MLlib K-Means API</td>
<td>25</td>
</tr>
<tr>
<td>4.2 Smart-MLlib Linear Regression API</td>
<td>31</td>
</tr>
<tr>
<td>4.3 Smart-MLlib Gaussian Mixture API</td>
<td>36</td>
</tr>
<tr>
<td>4.4 Smart-MLlib SVM API</td>
<td>41</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A simplified example dataflow in MapReduce</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>A simplified example dataflow in Spark</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>An iterative dataflow in Smart</td>
<td>10</td>
</tr>
<tr>
<td>3.1</td>
<td>The dataflow of Smart-MLlib when the API is used by a driver program</td>
<td>16</td>
</tr>
<tr>
<td>4.1</td>
<td>Comparison of k-means API using Smart-MLlib and Spark’s MLlib</td>
<td>26</td>
</tr>
<tr>
<td>4.2</td>
<td>Comparison of the linear regression API using Smart-MLlib and Spark’s MLlib</td>
<td>32</td>
</tr>
<tr>
<td>4.3</td>
<td>Comparison of the Gaussian mixture model API using Smart-MLlib and Spark’s MLlib</td>
<td>37</td>
</tr>
<tr>
<td>4.4</td>
<td>Comparison of the SVM API using Smart-MLlib and Spark’s MLlib</td>
<td>42</td>
</tr>
<tr>
<td>5.1</td>
<td>Performance comparison of k-means on Smart-MLlib and Spark’s MLlib</td>
<td>47</td>
</tr>
<tr>
<td>5.2</td>
<td>Comparison of scalability for k-means on Smart-MLlib and Spark’s MLlib</td>
<td>48</td>
</tr>
<tr>
<td>5.3</td>
<td>Performance comparison of linear regression on Smart-MLlib and Spark’s MLlib</td>
<td>50</td>
</tr>
<tr>
<td>5.4</td>
<td>Comparison of scalability for linear regression on Smart-MLlib and Spark’s MLlib</td>
<td>51</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.5</td>
<td>Performance comparison of the Gaussian mixture model algorithm on</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>Smart-MLlib and Spark’s MLlib</td>
<td></td>
</tr>
<tr>
<td>5.6</td>
<td>Comparison of scalability for Gaussian mixture model algorithm on</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>Smart-MLlib and Spark’s MLlib</td>
<td></td>
</tr>
<tr>
<td>5.7</td>
<td>Performance comparison of SVM on Smart-MLlib and Spark’s MLlib</td>
<td>55</td>
</tr>
<tr>
<td>5.8</td>
<td>Comparison of scalability for SVM on Smart-MLlib and Spark’s MLlib</td>
<td>56</td>
</tr>
</tbody>
</table>
## List of Listings

3.1 Import statements needed to access Smart-MLlib’s linear regression API 19
3.2 Example of a driver program using Smart-MLlib’s linear regression API 19
3.3 Example of Smart-MLlib’s linear regression API . . . . . . . . . . . . 21
4.1 Pseudocode for Smart’s k-means implementation . . . . . . . . . . . 24
4.2 Pseudocode for Smart’s linear regression implementation . . . . . . 29
4.3 Pseudocode for Smart’s Gaussian mixture model implementation . . 34
4.4 Pseudocode for Smart’s SVM implementation . . . . . . . . . . . . . 39
Chapter 1: Introduction

Big-data analytics, and the frameworks that support it, are becoming continuously more important in today’s world. Corporations, government agencies, and individuals are increasingly turning to big-data analytics to guide their decision making processes [26, 8]. The trend of data-driven decision making has created a new multi-billion dollar big-data technology and services industry [11]. At the heart of this industry are the numerous distributed-computing technologies that make computation feasible on a massive scale.

Since it was introduced in 2004, one of the most popular technologies used in distributed-computing has been Google’s MapReduce [4]. This programming model offers users an easy-to-use paradigm that can be utilized to solve a wide variety of problems. Due to its flexible nature and simple application, implementations of MapReduce are still popular in both industry and academia [16].

Although MapReduce has been very successful, the programming model is focused around linear dataflows. As the demands of big-data analytics grew to include iterative machine-learning applications, a new type of distributed-computing framework was needed. Apache Spark [1] was proposed in 2010 to address this need. Spark not only boasts an easy-to-use Scala interface, but also has built-in support for cyclic dataflows [30]. The notable benefits of Spark have resulted in a surge in its popularity
over the past few years. In 2015 alone, Spark received a $300 million investment from IBM and became the most active project, in terms of number of contributors, in the Apache Software Foundation [9].

1.1 Motivation

In part to demonstrate the support for cyclic dataflow in Spark, its makers implemented a full machine-learning library, coined MLlib, on top of the framework. This library not only provides concrete examples of iterative applications in Spark, but also provides an extremely user-friendly API for production-quality, machine-learning algorithms. For many of the algorithms, users need less than 20 lines of code to build complex statistical models from stored semi-structured data [24].

Although Spark was designed with iterative algorithms in mind [30], it is unclear whether Spark is the ideal system for a machine-learning specific library. Other distributed-computing frameworks, such as our in-Situ MapReduce liTe (Smart) system, have been shown to outperform Spark by at least an order of magnitude for some machine-learning tasks [28]. Building a machine-learning library that mimics Spark’s MLlib on top of Smart could lead to increased performance without compromising the easy-to-use interface.

1.2 Our Contributions

In this thesis, we present a machine-learning library prototype, comparable to Spark’s MLlib, built on top of our Smart system. As a demonstration of feasibility, this “Smart-MLlib” currently consists of four machine-learning algorithms: k-means clustering, linear regression, Gaussian mixture models, and support vector machines.
Each algorithm has been implemented in Smart and has a Spark’s MLlib-inspired Scala interface that is used to run the Smart application.

In addition to presenting the specific Smart-MLlib applications in this paper, we will also describe the underlying architecture of our system. The main focus of our architectural discussion revolves around launching Smart’s native jobs from within Scala’s Java virtual machine (JVM) environment. After describing the complications surrounding this issue, we explain why utilizing the scala.sys.process package and a single intermediate file remains the best way to communicate between Scala and Smart.

Beyond introducing Smart-MLlib and its architecture, we also detail results from testing our system against Spark’s MLlib. Through experimentation with inputs ranging from 1GB to 16GB and clusters ranging from 4 nodes to 32 nodes, we show that the speed of Smart-MLlib’s implementations outperform Spark’s MLlib implementations for every tested configuration. Specifically, Smart-MLlib outperforms Spark’s MLlib by a minimum of 90% for every configuration tested and an average of over 800% across all experiments. In addition to these performance results, we also show that Smart-MLlib scales better than Spark’s MLlib in almost all cases. On average, Smart-MLlib scales from 4 nodes to 32 nodes between 90% and 220% better than Spark’s MLlib for every algorithm tested.

1.3 Organization

This thesis is organized in the following way: Chapter 2 discusses the background and motivation for MapReduce, Spark, Smart, and MLlib. Chapter 3 introduces the
architecture used in our Smart-MLlib. Chapter 4 covers each of Smart-MLlib’s algorithms by giving a brief overview, exploring the Smart implementation, presenting the Smart-MLlib API, and comparing the usage of Smart-MLlib to the usage of Spark’s MLlib for the same algorithm. Chapter 5 presents and analyzes the experiments that were conducted to compare the libraries. Finally, Chapter 6 concludes the thesis with an overview of what was covered and provides topics for further work.
Chapter 2: Background

In this section, we provide background on all of the major technologies discussed within this report. First, we cover the history and basic ideas of the MapReduce programming model. Next, we discuss the motivation and basic theory of Spark. Third, we give an overview of Smart, our distributed-computing framework. Finally, we highlight the benefits of Spark’s machine-learning library, MLlib.

2.1 MapReduce

In the early 2000s, distributed computing was already making a large impact on the technology industry [2]. The ability to leverage clusters of computers to speed up computation was motivating Google and many other companies to implement hundreds of special-purpose distributed applications [4]. While this approach of creating custom applications on a per-problem basis was working, it was far from efficient. Straightforward algorithms were obfuscated by the details required to build distributed algorithms: explicit parallelization of the computation, intelligent distribution of the data, etc.

In an effort to both improve the accessibility of distributed computing and to simplify distributed-application code, Google released the MapReduce programming model in 2004 [4]. The model provides a very simple API containing only two core
functions: \textit{map()} and \textit{reduce()}. By implementing these functions, users are able to write applications which are automatically capable of operating on massively distributed systems. This means that the complex details inherent to distributed applications are hidden from the programmer.

![Diagram of MapReduce dataflow](image)

**Figure 2.1:** A simplified example dataflow in MapReduce

Figure 2.1 shows the simplified dataflow of MapReduce: First, \((\text{input\_key}, \text{value})\) pairs are provided to the \textit{map()} function. \textit{map()}, which is implemented by the user, consumes these pairs and returns a set of \((\text{intermediate\_key}, \text{value})\) pairs. MapReduce’s “shuffle” phase groups \((\text{intermediate\_key}, \text{value})\) pairs by their keys into \((\text{intermediate\_key}, \text{set of values})\) pairs which are passed to the \textit{reduce()} function. \textit{reduce()},
which is also implemented by the user, receives the \((\text{intermediate\_key}, \text{set of values})\) pairs, reduces them in some way, and emits the final set of \((\text{output\_key}, \text{value})\) pairs.

This simple dataflow, coupled with MapReduce’s straightforward, functional-style API, has made the programming model very popular for a variety of applications [7, 5]. Despite its success, the MapReduce implementations have had performance issues for certain types of algorithms. In particular, iterative algorithms do not perform well within MapReduce’s linear programming model [30].

2.2 Spark

Spark was introduced in 2010 to fulfill the need for a general-purpose, parallel-processing framework with built-in support for nonlinear dataflows [30]. Sticking to the MapReduce style, Spark used the Scala [21] programming language to provide users with a friendly, functional programming feel. Spark specifically addressed two use cases that MapReduce failed to adequately address: iterative jobs and interactive analytics [30]. In order to be performant when handling both of these tasks, Spark needed to be capable of holding a working set of data in memory within a distributed environment.

To address this challenge, Spark used a new dataset abstraction called resilient distributed datasets (RDDs). As defined in its original paper, a RDD is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost [30]. These RDDs gave Spark a distributed and fault-tolerant way to pull massive amounts of data into memory. In addition to defining these new distributed datasets, Spark defined a series of operations on RDDs that supported parallel computation.
RDD operations can be loosely grouped into two categories: transformations and actions [25]. Transformations take an RDD of one type, A, and transform it into an RDD of another type, B, using a user-defined function. Examples of transformations include `map()`, `flatMap()`, and `filter()`. Actions, on the other hand, require an actual computation to be performed. Actions process a particular RDD and produce some type of result. Examples of actions include `reduce()` and `collect()`. Both transformations and actions are performed in parallel by Spark.

![Figure 2.2: A simplified example dataflow in Spark](image)

Spark applications create RDDs, guide them through a series of transformations, and often terminate after performing some type of action. Figure 2.2 shows this dataflow in Spark. First, data is loaded from the file system into an RDD. After being loaded, a series of transformations are performed on the RDD. Finally, an action is performed and the program is terminated.

As Spark can clearly mimic MapReduce’s dataflow by performing a single RDD map transformation followed by a reduce action, it offers a clear improvement of MapReduce. This simple dataflow that seamlessly supports iterative applications has led to Spark becoming a frontrunner in distributed, machine-learning processing [9].
If current trends persist, Spark will only continue to gain market share in the coming years.

### 2.3 Smart

Smart [28] is the next generation of a parallel-computing framework that has evolved from FREERIDE (FRamework for Rapid Implementation of Data Mining Engines) [13] and MATE (Map-reduce with an AlternaTE API) [12]. All of these frameworks expose APIs that are similar to MapReduce’s API; however, it should be noted that the first paper on the FREERIDE system was published in 2001 [13] and, thus, predates Dean and Ghemawat’s MapReduce paper by three years.

While Smart’s API is similar to MapReduce’s API in many ways, the Smart system processes data in a substantially different way. Instead of map and reduce phases of computation, Smart uses reduction and combination phases. Both of these phases are supported by two map data structures: a combination map and a reduction map. These data structures are composed of user-defined reduction objects that store the accumulated information of relevant input.

More specifically, Smart processes data in the following way. First, the reduction phase takes place. Smart receives a chunk of data and maps it to a specific key. With this key, Smart locates a reduction object in the runtime’s reduction map. The chunk of data is then accumulated (or reduced) into the reduction object specified by the key. After all the data has been reduced, the combination phase begins. First, all reduction maps on a node are merged together locally into a single combination map. After the local merge, all of the combination maps are further merged into a final combination map on the master node.
Smart supports iterative algorithms by distributing the final combination map from the master node to every Smart instance between each pass through the data. The information contained in the map is then available to be referenced throughout every data-processing stage. Supporting this flow required the addition of a post-combination procedure that allows users to make any final updates to the master combination map before it is distributed. Furthermore, Spark provides functions to initialize the combination map before the first iteration and to convert the combination map to an output result after the last iteration.

Figure 2.3: An iterative dataflow in Smart

To support all of the functionality described above, Smart provides an API to the user. The core functions of the API can be seen in Table 2.1 and have been pulled from Smart’s official user guide [27]. Functions `gen_key()` and `gen_keys()` map
data chunks to specific reduction objects within the local reduction map. The `accumulate()` function reduces each data chunk into the reduction objects specified by `gen_key/s()`. After every data chunk has been reduced, `merge()` is used to combine all of the reduction maps into a single combination map. Before the combination map is distributed to each Smart instance, `post_combine()` can be used to perform any extra processing. Figure 2.3 shows this dataflow graphically. Two functions not shown are `process_extra_data()` which initializes the combination map and `convert()` which is used to convert the combination map to an output result.

<table>
<thead>
<tr>
<th>Smart’s User API</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Table 2.1:</strong> Smart’s main API functions</td>
</tr>
<tr>
<td><code>virtual int gen_key(const Chunk&amp; chunk, const In* data, const map&lt;int, unique_ptr&lt;RedObj&gt;&gt;&amp; com_map) const</code></td>
</tr>
<tr>
<td><code>virtual void gen_keys(const Chunk&amp; chunk, const In* data, vector&lt;int&gt;&amp; keys, const map&lt;int, unique_ptr&lt;RedObj&gt;&gt;&amp; com_map) const</code></td>
</tr>
<tr>
<td><code>virtual void accumulate(const Chunk&amp; chunk, const In* data, unique_ptr&lt;RedObj&gt;&amp; red_obj) = 0</code></td>
</tr>
<tr>
<td><code>virtual void merge(const RedObj&amp; red_obj, unique_ptr&lt;RedObj&gt;&amp; com_obj) = 0</code></td>
</tr>
<tr>
<td><code>virtual void process_extra_data(const void* extra_data, map&lt;int, unique_ptr&lt;RedObj&gt;&gt;&amp; com_map)</code></td>
</tr>
<tr>
<td><code>virtual void post_combine(map&lt;int, unique_ptr&lt;RedObj&gt;&gt;&amp; com_map)</code></td>
</tr>
<tr>
<td><code>virtual void convert(const RedObj&amp; red_obj, Out* out) const</code></td>
</tr>
</tbody>
</table>
By defining a reduction object and implementing the functions discussed above, Smart users are able to run massively distributed applications without explicitly dealing with any of the complexities of parallel programming. Moreover, Smart’s built-in support for cyclic dataflows makes it a natural choice for iterative applications. Together, these advantages make Smart an excellent framework for distributed machine-learning tasks.

2.4 MLlib

Spark is designed to support cyclic dataflows and, consequently, is a great framework to use for iterative algorithms [17, 15]. As many machine-learning algorithms contain iterative components, Spark has a distinct advantage with these problems. To highlight this advantage to new users, machine-learning examples have been available for Spark since its first release. In fact, Spark’s initial 2010 paper contains examples for both logistic regression and alternating least squares – two machine-learning algorithms.

While the provided machine-learning algorithm examples were great introduction programs for learning Spark, they did not provide commercial-level implementations to consumers. If users wanted finely tuned versions of these algorithms for production, they would need to implement the modified versions themselves. Although Spark prides itself on being an easy-to-use framework, implementing these sophisticated machine-learning algorithms is ambitious for many users.

To relieve users of the need to create their own commonly used machine-learning implementations, MLlib was developed in 2013 [17]. MLlib is a production-ready
distributed machine-learning library developed on top of Spark. Since Spark’s 0.8 release, MLlib has come packaged with Spark and has dramatically simplified machine-learning for Spark users. Through utilization of MLlib, users can easily load data into Spark, build models of the data with optimized machine-learning algorithms, and then query those models to extract meaning from the data. Amazingly, all three of these steps can commonly be implemented in less than 30 lines of Scala code.

As of the 1.6.1 release, MLlib contains algorithms for many common machine-learning tasks including classification, regression, collaborative filtering, clustering, and dimensionality reduction [24]. Beyond these conventional machine-learning tasks, MLlib also contains modules for basic statistical calculations. These statistics modules perform tasks ranging from a simple mean calculation to hypothesis testing and random data generation.

Needless to say, MLlib is a very useful library. It has successfully emphasized Spark’s ability to gracefully handle iterative algorithms while providing tremendous utility to the big-data community. Users are automatically handed dozens of production-ready, machine-learning algorithms via MLlib, and this improved accessibility has undoubtedly contributed to the continued success of Spark.
Chapter 3: Architecture

Recall that this thesis presents a high-performance, machine-learning library comparable to Spark’s MLlib and built on top of our Smart system. This chapter focuses on the architecture we used to create this library prototype. First, we give an architectural overview that includes the rationale for our approach. After introducing the system, we enumerate possible advantages and disadvantages resulting from our design choices. Finally, the low-level details of the architecture are demonstrated through a programmatic walk-through of what occurs when the library is utilized.

3.1 System Overview

At its core, Smart-MLlib is a Scala-based API that is used to execute machine-learning algorithms on Smart. Scala is a flexible, high-level language that – most importantly for our discussion – is executed within the Java virtual machine (JVM). Smart, on the other hand, is written in C++ with parallelization handled by a combination of OpenMP [3] and MPI [6]. The chief architectural decisions made while developing our system revolved around connecting a Scala API, which is called from within the JVM, to the natively run Smart system.

Connecting a JVM-based language, like Scala, with a native language, like C++, is not a new problem. In fact, there is a common interface called the Java Native
Interface (JNI) [14] which is regularly used for this purpose. By using JNI, native code can be imported directly into JVM-based languages. This allows a C++ function to be seamlessly called from within an executing Scala program.

While the JNI appears to be a perfect fit for our machine-learning library, there is a substantial barrier to using this technology. Smart is not just a C++ library, but a C++ library that uses MPI to distribute work across clusters of nodes. MPI requires a runtime environment to be initialized through the use of a `mpirun` or `mpiexec` command. While it is theoretically possible to set up this runtime environment programmatically within a C or C++ program, it is not recommended.

Due to the complexity of initializing the MPI runtime environment from within C or C++, we decided not to use JNI to perform the communication between our Scala API and Smart. Instead, we determined the best way to communicate between Scala and Smart was through the `scala.sys.process` package [22]. This package allows us to conveniently launch a Smart job from within Scala using the `mpiexec` command. In doing so, the package removes all of the complexity that surrounds initializing the MPI runtime environment.

After deciding on using `scala.sys.process` as the communication method between the Scala API and Smart, the rest of the system design fell into place. The overall flow of the system, which is shown pictorially by Figure 3.1, is as follows:

1. The Scala API is called by the user.

2. The Scala API prepares a `mpiexec` command complete with all arguments needed by Smart.
3. Using `scala.sys.process`, the Smart job is executed with the `mpiexec` command prepared in Step 2. The Scala process blocks until the Smart job terminates.

4. The Smart job executes the desired machine-learning algorithm.

5. Before finishing, the Smart job writes the result of Step 4 (as a model) to disk.

6. The Smart job finishes and the Scala process unblocks.

7. The model produced by Step 5 is immediately read into a Scala object.

8. The Scala API returns the model from Step 7 to the user as a Scala object.
3.1.1 System Advantages

Architecting the system in the ways outlined above provides several advantages
over other possible designs. First, calling Smart as an external command, as it would
be called without the MLlib wrapper, ensures that all necessary runtime configura-
tions are properly set up. The assurance of proper runtime initialization is particularly
convenient, since Smart leverages OpenMP and MPI for parallelization which makes
manual configuration of the runtime environment very complex.

Beyond the simplification of launching Smart jobs, the architecture also forces
every MLlib algorithm to define a savable model. As shown in Step 5 of Figure 3.1,
this requirement comes from the Smart executable, which finishes only after saving
a model to disk. Having a savable model has clear benefits in big-data processing.
When jobs can take hours or even days to complete, the ability to save the model
produced from these jobs can save a lot of time. Additionally, since the model is
saved to persistent storage, if the JVM process crashes after a Smart job terminates,
the machine-learning job doesn’t need to be executed again.

3.1.2 System Disadvantages

While the system architecture utilized has several benefits, it also introduces a few
disadvantages. The main downside is the system’s lack of control over the execution of
a Smart application. Since Smart is called as an external process, exception handling
and graceful recoveries are very difficult to achieve. The only real usable information
Scala gets directly from Smart is the exit status after the Smart job finishes execution.

In addition to the loss of control over the execution of Smart jobs, the architecture
also brings additional latency into the system. Writing and reading from disk is a
very slow way to communicate between two processes. That being said, the additional
time spent writing and reading a small file to disk is miniscule when compared with
the total execution time of the Smart job.

Finally, the system architecture does introduce an additional point of failure into
the system. Writing and reading from disk can cause failures, and because the writing
and reading occurs on different processes, it is sometimes difficult to recover from and
detect these failures. Fortunately, these failures are infrequent, and many of the issues
that do appear can easily be handled in the language in which they occur.

3.2 Smart-MLlib Walkthrough

In Section 3.1, an overview of the system architecture was presented. Here, we
will take a closer look at each individual step in the dataflow. To help visualize the
data movements, sample Scala code will be presented throughout the description.
While the code being used is specific to the linear regression algorithm, the dataflow
described is common to all implementations. Since understanding linear regression is
not necessary for this section, we will not be describing the algorithm here. For more
information on the algorithm, please refer to Section 4.2.

As is the case with any API, Smart-MLlib’s lifecycle begins when the library is
called from within a Scala program. In order to reference the correct API, the external
program is required to import the relevant classes. Each algorithm implemented in
Smart-MLlib requires two classes — one responsible for execution of the machine-
learning algorithm and one responsible for result storage of the algorithm. For linear
regression, the import statements are shown in Listing 3.1.
Listing 3.1: Import statements needed to access Smart-MLlib’s linear regression API

```java
import edu.osu.smart.mllib.regression.LinearRegression;
import edu.osu.smart.mllib.regression.LinearRegressionModel;
```

After importing the proper libraries, the users can start writing their applications.

The main tasks for which these applications are responsible are as follows:

1. Preparing the arguments for the API call

2. Calling the API with the arguments prepared in Step 1

3. Extracting the relevant data from the model returned in Step 2

Listing 3.2 shows all three of these steps carried out programmatically for the sample linear regression program: Lines 1 - 16 prepare all of the necessary data for the linear regression API, lines 18 - 28 actually perform the API call, and lines 30 - 35 extract information from the model returned by the API.

Listing 3.2: Example of a driver program using Smart-MLlib’s linear regression API

```java
// 1. Preparing the arguments for the API call
val dataPath = "my_kmeans_data.h5";
val dataVarName = "point"
val sampleDimensions = 16
val learningRate = 0.2
val hasBias = true
val its = 1000
val modelOutputFile = "temp_linear_regression_model.m"
```

// 2. Calling the API with the arguments prepared in step 1

// 3. Extracting the relevant data from the model returned in step 2
Since the logic of the machine-learning algorithms is contained in Smart, as opposed to Scala, the actual Scala API function is remarkably simple. In particular, the API is only responsible for the following tasks:

1. Creating the external command used to execute the Smart job (i.e. an `mpiexec` command)

2. Executing the Smart job

3. Reading the model produced by Smart into a Scala Object

4. Deleting the model file created by Smart (optional)

5. Returning the model from Step 3 to the caller

A programmatic example of a Smart-MLlib API function is shown in Listing 3.3. Clearly, all five of the steps enumerated above are explicitly handled in the provided Scala code.
Listing 3.3: Example of Smart-MLlib’s linear regression API.

```scala
// 1. Creating command from arguments
val commandString = generateCommand(
    nativeExeContext, dataPath, dataVarName,
    sampleDimensions, learningRate, hasBias,
    iterations, modelOutputFile
)

// 2. Executing command and blocking until finished
val command = Process(commandString)
val exitCode = command !

// 3. Reading file created by command into LR Model
val model =
    buildModelFromFile(modelOutputFile, sampleDimensions)

// 4. Deleting temp file
new File(modelOutputFile).delete()

// 5. Returning model
return model
```

Once the model is returned from the API, the dataflow of our system is finished. Note that while Smart is used for the actual logic of our machine-learning library, the details of the Smart implementation have been omitted from this section. As far as the architecture of Smart-MLlib is concerned, Smart is simply a compiled executable that can be run using the `scala.sys.process` package.
Chapter 4: Smart-MLlib Implementation

After focusing on the architecture of our machine-learning library prototype in Chapter 3, we now concentrate on the various applications developed for the system. In order to establish the feasibility of Smart-MLlib, we implemented four machine-learning algorithms for the library that are also available in Spark’s MLlib: k-means clustering, linear regression, Gaussian mixture models, and support vector machines. This chapter presents each of these algorithms in the following way: 1) the algorithm will be introduced with a brief overview; 2) the Smart implementation of the algorithm will be covered; 3) the algorithm’s Smart-MLlib interface will be introduced; and 4) the interface will be compared with the interface provided by Spark’s MLlib. After presenting all of the algorithms individually, the section is concluded with an overview of how the two APIs compare.

4.1 K-Means Clustering

The first algorithm to be covered is k-means clustering (k-means). K-means is an unsupervised machine-learning technique used to separate a dataset into $k$ groups such that each group contains similar patterns [23]. K-means, in particular, typically uses the Euclidian distance between two patterns as the measure of similarity [10]. The goal of the algorithm is to group all of the patterns into $k$ clusters in such a way
that the sum of the squared distance between every data pattern and its assigned
cluster center is minimized [23].

The basic k-means algorithm works as follows: 1) the initial $k$ centers are set;
2) each data point in the dataset is assigned to the nearest center; 3) each center
recomputes its location as the mean of all data points assigned to it; and 4) Steps
2 and 3 are repeated until a stopping condition is met. For a more comprehensive
explanation of the k-means algorithm, please refer to Spark’s MLlib guide [24].

4.1.1 Smart’s Implementation

To fully describe Smart’s k-means implementation, both the reduction object and
core API functions need to be defined. The pseudocode for these items can be seen
in Listing 4.1 which was adapted from Smart’s original paper [28]. The reduction
object, $ClusterObj$, represents a single cluster center. It has two main roles: 1) to
maintain the current location of the cluster’s center and 2) to accumulate all of the
input vectors assigned to the cluster.

The implemented functions process every data chunk as an input point. First,
$gen\_key()$ maps each input point to the nearest $ClusterObj$ in the combination map
and returns its key. Second, $accumulate()$ is used to accumulate $sum$ and $size$ on the
reduction object specified by $gen\_key()$. Next, $merge()$ accumulates all the reduction
maps produced by $accumulate()$ into a single combination map. The final combination
map, which holds one $ClusterObj$ for each cluster in the algorithm, is then updated
by $post\_combine()$. The update uses the accumulated $sum$ and $size$ to move each
$ClusterObj$’s centroid to the mean of the data points assigned to it. Finally, $sum$ and
$size$ are reset and the process repeats with the newly calculated combination map.
Listing 4.1: Pseudocode for Smart’s k-means implementation

```cpp
struct ClusterObj : public RedObj {
    T centroid[NUM_DIMS];
    T sum[NUM_DIMS];
    size_t size = 0;

    // Update centroid based on sum and size
    void update_centroid();
};

class Kmeans : public Scheduler<T, T*> {
    // combination_map is a set of clusters (i.e. ClusterObj)
    // chunk represents the next data point being considered

    // Return nearest cluster center key to the input
    int gen_key(chunk, data, combination_map) {
        /* 1. Find nearest cluster, c, in combination_map to the
           data point represented by chunk. */
        /* 2. Return the key associated with cluster c. */
    }

    // Accumulate sum and size.
    void accumulate(chunk, data, red_obj) {
        red_obj->sum += chunk; // vector addition
        red_obj->size++;
    }

    // Merge sum and size.
    void merge(red_obj, com_obj) {
        com_obj->sum += red_obj->sum; // vector addition
        com_obj->size += red_obj->size;
    }

    // Set up the initial centroids in combination_map.
    void process_extra_data(extra_data, combination_map) {
        /* 1. Transform extra_data into set ClusterObjs C */
        /* 2. Load C into the combination map */
    }

    // Update the clusters for the next iteration
    void post_combine(combination_map) {
        for (/* each ClusterObj in combination_map*/) {
            ClusterObj.update_centroid();
            /* Clear sum and size in ClusterObj */
        }
    }
};
```
4.1.2 Smart-MLlib Interface

<table>
<thead>
<tr>
<th>API Functions</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>API Parameters</th>
<th>Parameter Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nativeExeContext</td>
<td>A wrapper for the MPI/OpenMP information needed to run the Smart executable.</td>
</tr>
<tr>
<td>dataPath</td>
<td>The location of the data file that will be processed.</td>
</tr>
<tr>
<td>dataVarName</td>
<td>The variable type of the data to be processed (only relevant to scientific formats e.g. netcdf).</td>
</tr>
<tr>
<td>dimensions</td>
<td>The dimensionality of the data to be processed.</td>
</tr>
<tr>
<td>numCentroids</td>
<td>The number of clusters to create for the k-means model.</td>
</tr>
<tr>
<td>iterations</td>
<td>The number of iterations for which to run the k-means algorithm.</td>
</tr>
<tr>
<td>initialKMeansModel</td>
<td>The initial model (i.e. cluster centers) to use for the k-means algorithm.</td>
</tr>
<tr>
<td>modelOutputFile</td>
<td>The file used for the communication between Smart and Scala. Smart writes the model to this file and, subsequently, Scala reads the model from the file.</td>
</tr>
</tbody>
</table>

Table 4.1: Smart-MLlib K-Means API

The k-means API in Smart-MLlib is currently implemented as a single function. Table 4.1 gives the API and describes all of the formal parameters that may be specified. From examining the table, it is clear that the interface supplies options for declaring the initial k-means model, determining the number of clusters to use (i.e. k), and setting the number of iterations for the algorithm. The parameters unused for these tasks provide general information on the data being processed and the environment Smart is using to execute the distributed algorithm.
4.1.3 Comparison with Spark’s MLlib

The k-means API for Spark’s MLlib and Smart-MLlib provides very similar functionality for the basic k-means algorithm. Both interfaces allow the user to easily specify the number of clusters to create and the maximum number of iterations the algorithm should perform. In addition, as depicted in Figure 4.1, both APIs are used in a similar way.

Although the APIs are similar, one slight difference between the two libraries is that Spark’s MLlib implementation can pick the initial $k$ centers by randomly selecting $k$ points from the provided dataset; however, Smart-MLlib’s version requires the initial centers to be included in the API call. It should also be noted that Spark’s MLlib interface provides the ability to utilize a more sophisticated version of k-means called k-means∥ [24]. This modified algorithm uses a more intelligent method for
selecting the initial $k$ centers which can dramatically increase the algorithm’s convergence speed.

Beyond the additional k-means implementation, Spark’s MLlib API also provides the ability to specify a convergence condition. Once the convergence condition is met, the algorithm returns the k-means model and is not required to finish any remaining iterations. This feature is not yet supported by Smart and, thus, Smart-MLlib’s API does not provide an option for specifying a convergence condition.

4.2 Linear Regression

Linear regression is a technique used to model the relationship between variables [29]. In the version we discuss, the idea is to determine the linear combination of independent variables that will best explain a single dependent variable within a dataset. Therefore, for independent variables $x_1, x_2, ..., x_n$, and dependent variable $y$ in Equation 4.1, linear regression aims to select the constants $w_0, w_1, ..., w_n$ such that the relationship between the independent variables and dependent variable in the dataset is most accurately captured.

$$y = w_0 + w_1 * x_1 + ... + w_n * x_n$$ (4.1)

A common way to select these constants, and the method used in this paper, is through the least-mean-square algorithm which leverages gradient descent [10]. The basic algorithm proceeds as follows: 1) the initial weights are selected; 2) for each sample in the dataset, the weighted error between the actual dependent variable (from the sample) and the guessed dependent variable (from the sample’s independent variables and current values of $w_0, w_1, ..., w_n$) is summed; 3) $w_0, w_1, ..., w_n$ are updated
based on the sum of weighted errors computed in the previous step; and 4) Steps 2 and 3 are repeated until convergence. For a more comprehensive explanation of the linear regression algorithm, please refer to Spark’s MLlib guide [24].

4.2.1 Smart’s Implementation

The Smart implementation for linear regression can be fully described by defining both the reduction object and the core API functions for the algorithm. The pseudocode for the reduction object and the core API functions can be found in Listing 4.2. The reduction object in this algorithm, \textit{WeightObj}, represents the state of a full linear model. More specifically, a \textit{WeightObj} contains a set of weights that describe a linear function. In addition, \textit{WeightObj} is responsible for accumulating the number of points processed and the sum of weighted errors detected throughout the linear regression program.

Each data chunk being processed is viewed as an \textit{(output, input vector)} pair. As can be seen from the \textit{gen.key()} function, each of these input pairs is mapped to the same reduction object. This is because the linear model needs to process every point to perform the necessary weight update. In \textit{accumulate()}, the input pair is processed and both the \textit{size} and \textit{sum.weighted.error}, which is based on the input vector and current model weights, are reduced into the reduction object. \textit{merge()} further accumulates the reduction maps produced by \textit{accumulate()} into a single combination map. Once finalized, the combination map utilizes the \textit{size} and \textit{sum.weighted.error} attributes accumulated inside \textit{post.combine()} to perform a gradient-descent based update on its weights. Finally, the accumulating fields are cleared and the process restarts with the updated weights.
Listing 4.2: Pseudocode for Smart’s linear regression implementation

```cpp
struct WeightObj : public RedObj {
    // add 1 for possible bias term
    double weights[input_dims + 1];
    double sum_weighted_error[input_dims + 1];
    size_t size = 0;
    double learning_rate;
    bool has_bias;

    // Update the weights using gradient descent rule w/
    // sum_weighted_error, size, and learning_rate.
    void update_weights();
};

class LinearRegression : public Scheduler<T, T*> {
    // combination_map is a single WeightObj
    // chunk is the next sample to be processed:
    // chunk = <output, input_1, input_2, ...>
    // Let chunk.output = output
    // and chunk.input = <input_1, input_2, ...>

    // Only one reduction object, return a constant key
    int gen_key(chunk, data, combination_map) {
        return 0;
    }

    // Accumulate sum_weighted_error and size.
    void accumulate(chunk, data, red_obj) {
        double actual = red_obj.bias +
        chunk.input /  prod /  red_obj.weights;
        double desired = chunk.output;
        double error = desired - actual;
        // scalar-vector multiplication
        double weighted_error[input_dims + 1] =
            error * <1, chunk.input>;
        // vector-vector addition
        red_obj.sum_weighted_error += weighted_error;
        weight_obj=size++;
    }

    // Merge sum and size.
    void merge(red_obj, com_obj) {
        // vector addition
```
void process_extra_data(extra_data, combination_map) {
    /* 1. Transform extra_data into WeightObj W */
    /* 2. Load W into the combination map */
}

void post_combine(combination_map) {
    /* Get single WeightObj out of combination map */
    WeightObj.update_weights();
    /* Clear sum_weighted_error and size in WeightObj */
}

4.2.2 Smart-MLlib Interface

The Smart-MLlib linear regression API is shown in Table 4.2. The interface provides options for altering both the learning rate of the algorithm as well as the number of iterations the algorithm should perform. In addition to these parameters, the API also allows users to specify an initial model and gives an option for including a bias or intercept term in the linear-regression computation. The remaining parameters provide relevant information about the data being processed and the cluster environment Smart is using to execute the distributed algorithm.

4.2.3 Comparison with Spark’s MLlib

The linear regression API for both Smart-MLlib and Spark’s MLlib provides extremely similar functionality. Both APIs allow the user to modify the starting weights, the number of iterations, and the learning rate of the algorithm. In addition, both interfaces give an option to include or exclude a bias from the generated model.
API Functions

```python
def run( nativeExeContext: NativeExeContext, dataPath: String, dataVarName: String, sampleDimensions: Int, learningRate: Double, hasBias: Boolean, iterations: Integer, initialLRModel: LinearRegressionModel, modelOutputFile: String ): LinearRegressionModel
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nativeExeContext</td>
<td>A wrapper for the MPI/OpenMP information needed to run the Smart executable.</td>
</tr>
<tr>
<td>dataPath</td>
<td>The location of the data file that will be processed.</td>
</tr>
<tr>
<td>dataVarName</td>
<td>The variable type of the data in to be processed (only relevant to scientific formats e.g. netcdf).</td>
</tr>
<tr>
<td>sampleDimensions</td>
<td>The dimensionality of the samples to be processed (Output + Input).</td>
</tr>
<tr>
<td>learningRate</td>
<td>The learning rate or step size. Used to modify the effect of the gradient in weight updates.</td>
</tr>
<tr>
<td>hasBias</td>
<td>True iff the regression should factor in a bias/intercept term.</td>
</tr>
<tr>
<td>iterations</td>
<td>The number of iterations for which to run the algorithm.</td>
</tr>
<tr>
<td>initialLRModel</td>
<td>The initial model to use for the linear regression algorithm.</td>
</tr>
<tr>
<td>modelOutputFile</td>
<td>The file used for communication between Smart and Scala. Smart writes the model to this file and, subsequently, Scala reads the model from the file.</td>
</tr>
</tbody>
</table>

Table 4.2: Smart-MLlib Linear Regression API

Furthermore, the usage of Smart-MLlib’s API is nearly the same as the usage of the singleton-based API provided by Spark’s MLlib. Example usages of both these interfaces can be seen in Figure 4.2.

In addition to the singleton-based API, Spark’s MLlib also offers a class-based API shown in Figure 4.2c. The class-based interface has been included because it is necessary to use for some of the more advanced configurations of linear regression that Smart-MLlib supports. It is interesting to note that the singleton-based API is simply a wrapper for the class-based API.
val model = LinearRegression.run(nativeExeContext, dataPath, dataVarName, sampleDimensions, learningRate, hasBias, iterations, initialModel, modelOutputFile)

(a) Example usage of Smart-MLlib’s interface for the linear regression algorithm

val alg = new LinearRegressionWithSGD()
alg.setIntercept(true)
alg.optimizer.setStepSize(stepSize)
alg.optimizer.setNumIterations(iterations)
val model = alg.run(parsedData, initialWeights)

(c) Example usage of Spark’s MLlib interface (class-based) for the linear regression algorithm

Figure 4.2: Comparison of the linear regression API using Smart-MLlib and Spark’s MLlib

While the base functionality of both APIs are the same, a few differences are apparent. Most significantly, Spark’s MLlib supports the addition of a convergence criteria that, once met, terminates the algorithm without completing any further iterations. As previously noted, Smart is not yet able to support this feature and, thus, it has been omitted from the Smart-MLlib API. Another difference is the “miniBatch” option which Spark’s MLlib provides to allow users to select a random fraction of the input data for each iteration. This feature is also not yet supported by Smart. Lastly,
there are a couple advanced, gradient-descent configurations available for the class-based API provided by Spark’s MLlib, which are not yet available for Smart-MLlib.

### 4.3 Gaussian Mixture Model

A Gaussian mixture model (GMM) is a probability distribution that is constructed using a weighted combination of *k* Gaussian functions. The aim of training a GMM is to modify the weights (i.e. linear coefficients), means, and covariance matrices of the Gaussian functions in order to maximize the likelihood that a particular dataset could be generated by the mixture model [19]. Typically, a GMM is trained through the utilization of the expectation-maximization (EM) algorithm [19, 18].

Within the context of GMM, the EM algorithm works as follows: 1) the initial *k* Gaussians are selected; 2) the responsibility of each Gaussian to every data point is determined; 3) based on the responsibilities computed in the previous step, the Gaussian weights, means, and covariance matrices are updated; and 4) Steps 2 and 3 are repeated until convergence. For a more comprehensive explanation of the GMM algorithm, please refer to Spark’s MLlib guide [24].

#### 4.3.1 Smart’s Implementation

Smart’s GMM implementation can be described through defining the algorithm’s reduction object and core API functions. The pseudocode for both of these items can be seen in Listing 4.3. GMM’s reduction object, `GMMRedObj`, represents a full Gaussian mixture model. This means that it must maintain every Gaussian functions’ weight, mean, and covariance information. In addition to supplying these items, `GMMRedObj` is used for accumulating responsibility information for each Gaussian as the data is processed.
The logic of the data processing in this GMM implementation is very similar to the traditional EM algorithm but has been adapted to fit Smart’s alternative dataflow. Each data chunk, which is viewed as an individual data point, is first processed by the gen_key() function. Since this algorithm only requires a single reduction object, gen_key() simply returns a constant number. For each point processed, accumulate() computes and accumulates a wide range of responsibility information into a reduction object. After accumulate() finishes reducing the input, merge() combines all produced reduction maps into a single combination map. In post_combine(), this final combination map updates its Gaussian mixture model information based on the responsibilities accumulated. Finally, the accumulating data structures are cleared and the process repeats with the updated GMM information.

Listing 4.3: Pseudocode for Smart’s Gaussian mixture model implementation

```cpp
struct GMMRedObj : public RedObj {
    // These define the current mixture model
    double current_weights[K];
    double current_means[K][INPUT_DIMS];
    double current_covariances[K][INPUT_DIMS][INPUT_DIMS];
    // These 3 sums accumulate relevant 'responsibilities' that are used in the // expectation-maximization (EM) update
    double sum_gaussian_resp[K];
    double sum_input_resp[K][INPUT_DIMS];
    double sum_covariance_resp[K][INPUT_DIMS][INPUT_DIMS];
    // Returns the probability of input on multivariate // gaussian gaussianNum
    double inline mvnpdf(input, gaussianNum);
    // Updates current_weight, current_means, and // current_covariances using all of the responsibility // sums and the EM algorithm.
    void update_weights();
};

class GMM : public Scheduler<double, double**> {
    // Only one reduction object, return a constant key
```
int gen_key(chunk, data, combination_map) {
    return 0;
}

// Accumulate all responsibility information
void accumulate(chunk, data, red_obj) {
    for (/* each Gaussian g in red_obj */) {
        double resp = /* find responsibility of g for input */
        red_obj->sum_gaussian_resp[g] += resp;
        red_obj->sum_input_resp[g] += (resp * chunk);
        red_obj->sum_covariance_resp[g] +=
        /* resp * (chunk->current_means[g])^2 */
    }
}

// Merge all responsibility information
void merge(red_obj, com_obj) {
    for (/* each Gaussian g in red_obj */) {
        // range from scalar additions to matrix additions
        com_obj->sum_gaussian_resp[g] +=
        red_obj->sum_gaussian_resp[g];
        com_obj->sum_input_resp[g] +=
        red_obj->sum_input_resp[g];
        com_obj->sum_covariance_resp[g] +=
        red_obj->sum_covariance_resp[g];
    }
}

// Set up initial Gaussians in combination_map.
void process_extra_data(extra_data, combination_map) {
    /* 1. Transform extra_data into GMMRedObj G */
    /* 2. Load G into the combination map */
}

// Update Gaussians before next iteration
void post_combine(combination_map) {
    /* Get single GMMRedObj from combination map */
    GMMRedObj.update_weights();
    /* Clear accumulating data structures
     in GMMRedObj */
}
### API Functions

```python
def run( nativeExeContext: NativeExeContext, dataPath: String, dataVarName: String, numGaussians: Int, inputDimension: Int
iterations: Integer, initialGMModel: GaussianMixtureModel, modelOutputFile: String ): GaussianMixtureModel
```

### API Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nativeExeContext</td>
<td>A wrapper for the MPI/OpenMP information needed to run the Smart executable.</td>
</tr>
<tr>
<td>dataPath</td>
<td>The location of the data file that will be processed.</td>
</tr>
<tr>
<td>dataVarName</td>
<td>The variable type of the data to be processed (only relevant to scientific formats e.g. netcdf).</td>
</tr>
<tr>
<td>numGaussians</td>
<td>The number of Gaussians used in the GMM.</td>
</tr>
<tr>
<td>inputDimension</td>
<td>The dimensionality of the input.</td>
</tr>
<tr>
<td>iterations</td>
<td>The number of iterations for which to run the algorithm.</td>
</tr>
<tr>
<td>initialGMModel</td>
<td>The initial model to use for the gaussian mixture algorithm.</td>
</tr>
<tr>
<td>modelOutputFile</td>
<td>The file used for communication between Smart and Scala. Smart writes the model to this file and, subsequently, Scala reads the model from the file.</td>
</tr>
</tbody>
</table>

Table 4.3: Smart-MLlib Gaussian Mixture API

### 4.3.2 Smart-MLlib Interface

Since the Gaussian mixture model algorithm is similar in flavor to the k-means algorithm, the Smart-MLlib interface for these applications is almost identical. As Table 4.3 shows, the GMM interface provides options for specifying the initial model, the number of Gaussians to use, and the number of iterations for the algorithm to complete. The other parameters are used to specify both the data being processed and the system on which the Smart job will be running.
Figure 4.3: Comparison of the Gaussian mixture model API using Smart-MLlib and Spark’s MLlib

4.3.3 Comparison with Spark’s MLlib

The Gaussian mixture model APIs for Smart-MLlib and Spark’s MLlib provide nearly identical functionality. Both interfaces support setting the initial model, the number of Gaussians, and the maximum number of iterations. As mentioned in Sections 4.2.3 and 4.1.3, one difference between the APIs is that Spark’s MLlib includes an option for a convergence condition that stops the algorithm from continuing to iterate. This feature is not yet supported by Smart-MLlib.

Although the functionality for both APIs are nearly the same, the usage is slightly different. Spark’s MLlib does not provide a singleton-based API for Gaussian mixture
models, so the usage of the interface is typical of Spark’s other class-based interfaces.

To see an example of how the APIs differ, please refer to Figure 4.3.

4.4 Support Vector Machine

Support vector machines (SVMs) are used to classify data into two groups. Unlike various other types of classifiers that do not make determinations on the “goodness” of a classification (e.g. perceptrons [20]), SVMs attempt to optimally classify datasets [10]. The classification is defined with a hyperplane that separates the dataset into two classes. To find the optimal hyperplane, an objective function is optimized that rewards a large margin of separation between the hyperplane and the dataset and penalizes misclassifications [23, 10].

While complex SVMs exist, we will be focusing on two-class linear SVMs. The gradient-descent based algorithm we use to build these SVMs utilizes a hinge loss function and works as follows: 1) the initial weights for the model are selected; 2) the partial derivatives of the hinge loss function, with respect to the weights, are accumulated for each point; 3) based on the partial derivatives found in the previous step as well as the number of points and iteration number, the weights are updated with a gradient-descent rule; and 4) Steps 2 and 3 are repeated until convergence. For a more comprehensive explanation of the SVM algorithm, please refer to Spark’s MLlib guide [24].

4.4.1 Smart’s Implementation

The Smart implementation for SVM can be described by defining the algorithm’s reduction object and core API functions. These items can be found in Listing 4.4. The reduction object for this algorithm, \texttt{GradientObj}, uses an array of weights to represent
a complete linear model. In addition to providing this model, \textit{GradientObj} is also responsible for keeping track of the iteration number, the number of input chunks processed, and the sum of the hinge loss function’s partial derivatives throughout the execution of the algorithm.

Since this SVM is a binary classifier, each data chunk is interpreted as a \textit{(class, input)} pair. First, the input pair is sent to the \textit{gen\_key()} function. As this algorithm requires only one reduction object per reduction map, \textit{gen\_key()} always returns a constant number. In \textit{accumulate()}, the size and partial derivatives, which are based on the input chunk and the combination map’s current weights, are accumulated into a reduction object. After all the data has been accumulated, \textit{merge()} combines the reduction maps produced from \textit{accumulate()} into a single combination map. Using this master combination map, \textit{post\_combine()} performs a gradient-descent update on the weights of the model using all of the accumulated values. Finally, the accumulating data structures are cleared and the process is repeated with the updated weights.

Listing 4.4: Pseudocode for Smart’s SVM implementation

```cpp
struct GradientObj : public RedObj {
    // add 1 for possible bias term
    double weights[INPUT_DIMS + 1];
    double pDerivative[INPUT_DIMS + 1];
    size_t size = 0;
    size_t iter = 1;

    // Perform Gradient descend with partial
    // derivatives and L2 regularization
    void update_weights(regular\_param, learn\_rate);
};

template<class T>
class SVM : public Scheduler<T, T*> {
    // Only one reduction object, return a constant key
    int gen_key(chunk, data, combination\_map) {
        return 0;
    }
};
```
4.4.2 Smart-MLlib Interface

As Table 4.4 suggests, the Smart-MLlib SVM API provides options for specifying the initial SVM model, setting the number of iterations, and declaring if the SVM model being produced should include a bias term. In addition to these parameters,
the interface also gives users the opportunity to alter the algorithm’s learning rule through the `learningRate` and `regParam` parameters. As with the other Smart-MLlib interfaces discussed, the remaining parameters exist to provide information on the data being processed and the system being used to execute the SVM algorithm.
Figure 4.4: Comparison of the SVM API using Smart-MLlib and Spark’s MLlib

4.4.3 Comparison with Spark’s MLlib

The interfaces provided for the SVM algorithm have very similar functionality in both Smart-MLlib and Spark’s MLlib. Both APIs allow users to specify the number of iterations to run, the step size, the learning rate, and several other configurations relevant to SVMs. Moreover, as can be seen in Figure 4.4, the usage of the Smart-MLlib API is very close to the usage of the singleton-based API provided by Spark’s MLlib. That being said, Spark’s singleton-based API does not provide all of the
functionality that Smart-MLlib offers. For this reason, Spark’s full-featured, class-based API has also been included in Figure 4.4.

As was the case for linear regression – another gradient-descent based algorithm – Spark’s MLlib includes a few advanced options excluded from the Smart-MLlib API. The extra Spark’s MLlib options include a mini-batch fraction, a convergence criteria, and a few advanced customizations for the gradient-descent rule. Specifics on all additional parameters can be found by referencing Spark’s MLlib guide [24].

4.5 Summary of Comparison with Spark’s MLlib

For the algorithms discussed throughout Chapter 4, the API provided by Smart-MLlib closely resembles the API of Spark’s MLlib. From a functional perspective, both APIs give users access to the same core parameters which modify the execution of the algorithms in identical ways. Furthermore, as can be seen in Figure 4.1 and Figure 4.2, the usage of the Smart-MLlib API is very close to the usage of the singleton-based API provided by Spark’s MLlib.

Although the APIs are largely the same, Spark’s MLlib has a few novel features not yet supported by Smart. Some examples of these features include mini-batch fractions, a convergence criteria, and advanced customizations for learning rules. Clearly, these features available through Spark’s MLlib API are not available in the Smart-MLlib API. It should be noted, however, that these additional options do not affect the core mechanics of the algorithms provided by the API.

Altogether, in most circumstances the Smart-MLlib API should be easily capable of replacing Spark’s MLlib API for the algorithms discussed in this chapter. Smart’s library provides the same core functionality as Spark’s with a similar easy-to-use
interface. This should make switching between the two frameworks as simple as changing a few lines of code.
Chapter 5: Experimental Results

While Chapter 3 and Chapter 4 discussed the implementation details of our machine-learning library prototype, this section focuses on performance testing. To benchmark the system against the industry standard, we compare the performance and scalability of Smart-MLlib with the performance and scalability of Spark’s MLlib.

First, we describe the environment used to perform all of the experiments. Following this description, we compare the performance and scalability of each algorithm presented in Chapter 4: k-means clustering, linear regression, Gaussian mixture models, and support vector machines. After analyzing each algorithm individually, we finish by discussing some of the larger trends present across all of the experiments.

5.1 Environment

Our experiments were all conducted on the same homogeneous, multi-core computing cluster. Specifically, our tests were performed using 4, 8, 16, and 32 nodes configurations. Each node in the cluster uses two Intel(R) Xeon(R) E5630 processors and contains 12 GB of main memory. The processors each have a combined total of eight computing cores that run at a base frequency of 2.53GHz.

For the Smart based experiments, MPI (MPICH hydra version 3.1) was used to communicate between nodes and OpenMP (libgomp-4.4.7) was used for intra-node
communication. The Spark experiments, on the other hand, used Spark’s standalone cluster for communication. To keep the performance comparisons fair, all tests were conducted with one process per node and eight threads per process. This translates to Smart using one MPI process and eight OpenMP threads per node and Spark using one executor and eight executor cores per node. It should also be noted that all Spark tests use version 1.5.2 of both Spark and MLlib.

5.2 K-Means Clustering Experiments

For the k-means experiments, the performance of the basic k-means implementation was compared between Smart-MLlib and Spark’s MLlib. The algorithm was tested using input sizes of 1GB, 2GB, 4GB, and 16GB and computing clusters of 4, 8, 16, and 32 nodes. The performance reported for each configuration is an average of five independent trials. In all of the tests, k-means was run with four cluster centers for exactly 1000 iterations on 16-dimensional input. Since Spark will stop iterating when a default convergence condition is met, the source code was modified to ensure all iterations actually occurred.

5.2.1 Results

The results of all the k-means experiments can be seen in Figure 5.1. For every configuration tested, the Smart library outperformed Spark’s implementation by at least 150%. In the most dramatic case, a 1GB input file was processed by 32 nodes 17 times faster with the Smart-MLlib implementation than with the Spark’s MLlib version. In addition to outperforming Spark in head-to-head experiments, Smart-MLlib also out-scaled Spark’s MLlib. Figure 5.2 shows this by tracking Smart’s speedup over Spark while increasing the number of nodes. Clearly, as all three of the speedups
Figure 5.1: Performance comparison of k-means on Smart-MLlib and Spark’s MLlib

have positive slopes, Smart is becoming faster relative to Spark with each additional node. Averaging over all three input sizes tested in every node configuration, the Smart-MLlib k-means implementation scales 220% better than Spark’s between 4
nodes and 32 nodes. As these results are consistent with all other results within this chapter, please refer to Section 5.6 for a detailed analysis.

![Figure 5.2: Comparison of scalability for k-means on Smart-MLlib and Spark’s MLlib](image)

* Note: Smart speedup over Spark is calculated by dividing the execution time of the algorithm on Spark’s MLlib by the execution time of the algorithm on Smart-MLlib.

### 5.3 Linear Regression Experiments

For the linear regression experiments, the performance of the linear regression implementation on Smart-MLlib and Spark’s MLlib were compared. The tests included input sizes of 1GB, 2GB, 4GB, and 16GB and computing cluster sizes of 4, 8, 16, and 32 nodes. As in Section 5.2, the results reported for each configuration are an average of five independent trials. In all of the tests, the linear regression processed input with 15 dimensions and 1 output dimension and ran for exactly 1000 iterations.
Furthermore, each linear regression model included an intercept term resulting in a total of 16 weights trained by the algorithm. Note that the Spark source code had to be modified to guarantee all 1000 iterations of the algorithm were completed.

5.3.1 Results

The results of the linear regression experiments can be seen in Figure 5.3. From examining the graphs, it is clear that the Smart-MLlib implementation outperforms Spark’s in every configuration. More specifically, the Smart-MLlib version is at least twice as fast as Spark’s MLlib implementation in every experiment. In the most extreme case, Smart-MLlib’s linear regression performs 15 times faster than Spark’s. As with k-means, the Smart-MLlib version also scales better than Spark’s. Figure 5.4 shows this superior scaling graphically. As the number of nodes increases, so does Smart-MLlib’s performance relative to Spark’s. Averaging over all three input sizes tested in every node configuration, Smart-MLlib’s linear regression implementation scales 220% better than Spark’s between 4 nodes and 32 nodes. As these results are consistent with all other results in this chapter, please refer to Section 5.6 for a detailed analysis.

5.4 Gaussian Mixture Model Experiments

The Gaussian mixture model (GMM) experiments were conducted to compare the performance of GMM on Smart-MLlib and Spark’s MLlib. Tests were carried out using input sizes of 1GB, 2GB, 4GB, and 16GB and cluster sizes of 4, 8, 16, and 32 nodes. The results reported for each configuration are the average of five independent tests. Since GMM takes substantially longer to execute than the other algorithms covered, each trial was only run for 100 iterations using a four Gaussian
Figure 5.3: Performance comparison of linear regression on Smart-MLlib and Spark’s MLlib model. Furthermore, to mitigate the influence of different linear algebra libraries, an input dimensionality of two was used. Again, it should be noted that the Spark source code had to be modified to guarantee all iterations of the algorithm were completed.
5.4.1 Results

The Gaussian mixture model results can be seen in Figure 5.5. The charts in the figure show that the Smart-MLlib implementation greatly outperformed the Spark implementation for all tested configurations. Interestingly, Smart’s performance relative to Spark’s was much stronger for this algorithm than any of the other algorithms presented in the chapter. Results from k-means, linear regression, and SVM show Smart having roughly a 2 to 15 times advantage over Smart; however, for the GMM tests, this range balloons to a 13 to 54 times advantage.
(a) Gaussian mixture model results on a 4-node cluster  
(b) Gaussian mixture model results on an 8-node cluster  
(c) Gaussian mixture model results on a 16-node cluster  
(d) Gaussian mixture model results on a 32-node cluster

Figure 5.5: Performance comparison of the Gaussian mixture model algorithm on Smart-MLlib and Spark’s MLlib

Smart’s large advantage in this algorithm, beyond the factors explained in Section 5.6, could be a result of the complex nature of GMM. Spark achieves efficiency by caching intermediate results (i.e. RDDs) in memory and reusing them in every
iteration. In complex algorithms like GMM, Spark can be forced to remove cached RDDs to free up memory for execution. These RDDs are later recomputed, but performance suffers. This issue is not present in Smart since our system does not cache intermediate results.

Another interesting aspect of these results, not explained in Section 5.6, can be seen in Figure 5.6. In all other algorithms presented, the Smart-MLlib implementation scales better than Spark’s in all cases. This means that every line segment in the figures depicting scale comparisons have had exclusively positive slopes. In Figure 5.6, we see a single negatively sloping line segment for both input sizes plotted. Following the one negatively sloping segment within the 1GB input line and the one negatively sloping segment within the 2GB input line, both lines continue trending in a typical positive-slope fashion. Even though the Spark implementation out-scaled Smart’s in one situation, on average, Smart-MLlib’s GMM implementation scales from 4 nodes to 32 nodes 90% better than Spark’s.

The trend in Figure 5.6 seems to reinforce our view that Spark is having memory strain when executing the GMM algorithm. In both negative sloping line segments, the improved scalability occurred as the input size went from $\frac{1}{4}$ GB to $\frac{1}{8}$ GB per node. We suspect that this decrease in input size allowed each Spark executor to cache one or more additional RDDs resulting in significantly improved performance. Following this boost from additional memory, the scalability trend returned to one typical of the other algorithms studied in this chapter.
Figure 5.6: Comparison of scalability for Gaussian mixture model algorithm on Smart-MLlib and Spark’s MLlib
* Note: Smart speedup over Spark is calculated by dividing the execution time of the algorithm on Spark’s MLlib by the execution time of the algorithm on Smart-MLlib.

5.5 SVM Experiments

For the SVM experiments, the performance of the linear SVM implementation is compared between Smart-MLlib and Spark’s MLlib. Tests were conducted using input sizes of 1GB, 2GB, 4GB, and 16GB and cluster sizes of 4, 8, 16, and 32 nodes. As mentioned in other experiments, the result reported for each configuration is an average of five independent trials. The parameters used for the SVM tests closely mirror those used in the linear regression experiments. Each test ran for exactly 1000 iterations on samples with 15 input dimensions and 1 output dimension. Additionally, the SVM model always included an intercept term so a total of 16 weights were trained.
by the algorithm in each test. Note that the Spark source code had to be modified to guarantee all iterations of the algorithm were completed.

5.5.1 Results

(a) SVM results on a 4-node cluster  
(b) SVM results on a 8-node cluster  
(c) SVM results on a 16-node cluster  
(d) SVM results on a 32-node cluster

Figure 5.7: Performance comparison of SVM on Smart-MLlib and Spark’s MLlib
The results of the SVM experiments are shown in Figure 5.7. As with the previous algorithms, the Smart-MLlib SVM implementation outperformed the Spark’s MLlib SVM implementation in every configuration. More specifically, Smart-MLlib’s version ran 90% to 1100% faster than Spark’s. In addition to outperforming Spark, Smart-MLlib also out-scaled Spark’s MLlib. Figure 5.8 shows this graphically through the positively sloped line segments. As the number of nodes increases, Smart-MLlib’s SVM performance gets better relative to Spark’s MLlib. Averaging over all three input sizes tested in every node configuration, Smart-MLlib’s SVM implementation scaled 200% better than Spark’s between 4 nodes and 32 nodes. Since these results are consistent with the other results in this chapter, please refer to Section 5.6 for a detailed analysis.

![Figure 5.8: Comparison of scalability for SVM on Smart-MLlib and Spark’s MLlib](image)

* Note: Smart speedup over Spark is calculated by dividing the execution time of the algorithm on Spark’s MLlib by the execution time of the algorithm on Smart-MLlib.
5.6 Analysis and Discussion

All of the results presented in this chapter show that, for the algorithms discussed, Smart-MLlib performs strictly better than Spark’s MLlib. In every configuration tested, the Smart implementation performed at least 90% times better than the Spark implementation. Moreover, for the k-means, linear regression, and SVM tests, Smart-MLlib’s implementation performed an average of 380% better than Spark’s MLlib implementation. If the GMM results are included in that average, the performance multiple grows to 800%.

The performance advantages of Smart result from three key differences between Smart and Spark [28]. First, Spark produces a large amount of intermediate data after map operations that need to be grouped and reduced by the system. In comparison, Smart performs reductions directly into reduction objects which removes the extra data creation and need for grouping. Second, Spark applications create and store many immutable intermediate states (i.e. RDDs) throughout a program’s execution. On the other hand, Smart operations all occur directly on reduction maps that can be reused between iterations. Third, Spark relies heavily on network communication for transmitting information – even when that data is being transmitted to the same node. In contrast, Smart leverages the shared-memory environment on each node to reduce network traffic as much as possible.

In addition to outperforming Spark’s MLlib, the results of the experiments also show that Smart-MLlib scales better than Spark’s MLlib for the algorithms tested. In every experiment, with the exception of a special case discussed in Section 5.4.1, the more nodes added to the problem, the better Smart’s implementation performed relative to Spark’s. On average, the Smart-MLlib implementations scaled from 4
to 32 nodes about 2 to 3 times better than the equivalent applications in Spark’s MLlib. Interestingly, there were many instances in which Spark’s version performed worse when more nodes were added. Usually, Spark’s performance worsened when the amount of data being sent to each node dropped below about 128 MB. In these same situations, Smart was able to achieve a speedup by utilizing additional nodes.

The reason Smart scales better than Spark is probably very closely related to the reason it performs faster in general. As described previously, Smart’s reduction objects eliminate the need for data creation and grouping. Additionally, Smart focuses on minimizing network traffic through utilization of a shared-memory environment on each node. In contrast, Spark does require grouping and relies more heavily on the network for communication. These differences allow Smart to have lower overhead than Spark for additional nodes and make the system scale more efficiently.
Chapter 6: Conclusion and Future Work

The need for accessible and scalable machine-learning implementations is continuously growing. This thesis presents a machine-learning library prototype geared to address this need. Smart-MLlib provides an easy-to-use Scala API for distributed machine-learning algorithms that executes on top of Smart. These interfaces are modeled off of Spark’s MLlib API, but for all four of the algorithms tested – k-means clustering, linear regression, Gaussian mixture models, and support vector machines – the Smart library dramatically outperformed and out-scaled Spark’s version. Since the interfaces used by both libraries are so similar, this implies that users can achieve a performance boost by switching libraries without any real impact on developer effort.

Although the findings presented in this thesis look promising, Smart-MLlib is still in its infancy. Spark’s MLlib implements dozens of algorithms that have yet to be explored by Smart. To gain better understanding of how the two systems compare, more MLlib algorithms should be added to Smart-MLlib and performance tested. Furthermore, Smart’s functionalities must be expanded to support some of the features common in Spark’s MLlib implementations (e.g. a convergence criteria that terminates a program once met). These improvements to Smart will allow Smart-MLlib to truly match Spark’s MLlib in terms of functionality. Lastly, to seriously
contend with Spark’s MLlib, a fault-tolerant version of Smart must be developed as well. While previous versions of Smart, such as MATE [12], have implemented a fault-tolerant option, this option is not currently available for Smart.
Bibliography


