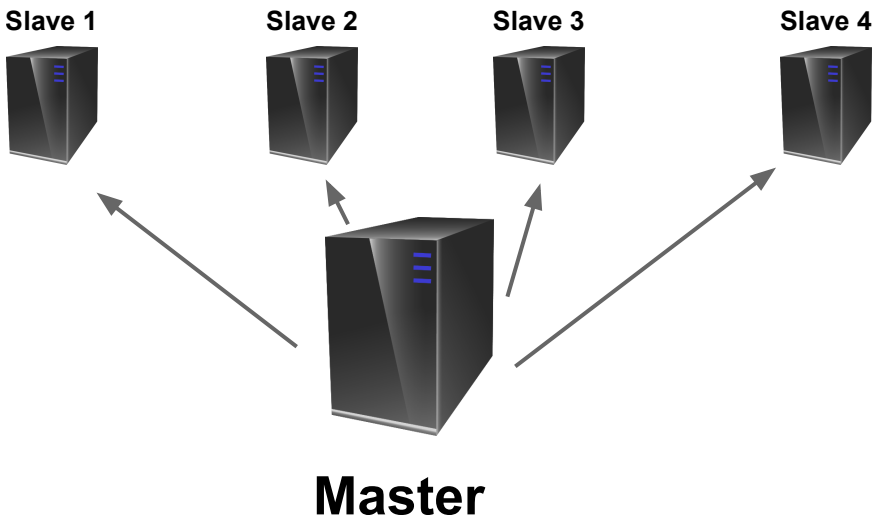

Figure 1: Master/ Slave Architecture

1 Cluster Architecture

To implement the iterative map-reduce [2] approach to ADMM, we use a traditional master-slave cluster architecture. Specifically, a single node, named the "master", coordinates the activities of a set of machines, called the "slaves", to perform independent tasks. This architecture is depicted in figure 1.

The rest of this section will discuss the particulars of this architecture in how it pertains to running the ADMM algorithm in master/slave manner.

1.1 Amazon EC2 and S3

Amazon provides developers with a set of web services that abstract away the difficulties of hardware and networking setup for deploying software. The first service we use is Amazon Elastic Cloud Compute (EC2). EC2 allows for temporary renting of computing nodes with IP addresses for *ssh*'ing into the machines. We use a full Linux environment on each one of our EC2 nodes. The master and all slaves are running different instance *types*. An instance type defines the specifications of the underlying hardware. The developer can choose the amount of RAM, hard disk space, cores, etc... We tested different instance types for the slave nodes to see what the performance benefits are for upgrading RAM and CPU. This is discussed in Section .

In addition, it is imperative for the running time of our experiments to have a fast data connection to download the data sets. Amazon provides a database service called Scalable Storage Service (S3). S3 gives the developer a set of buckets to store blobs under key values. Then, these keys can be converted into a publicly available web url. The master node on EC2 then downloads the

dataset from the S3 service. Since both EC2 and S3 reside within the same physical network, the data transfer speeds are much faster than if we had scp'ed the data directly from our own machines. An additional bonus is that Amazon allows data transfer from S3 to EC2 nodes free of charge.

1.2 Mesos Distributed Computing Platform

The Spark [6] software was developed with the Mesos [4] computing platform in mind. Mesos provides an abstraction level to developers that enables one to write distributed software without worrying about the underlying networking layer. Such a software package allows Spark to exist and to be written in a concise manner. This report does not discuss Mesos in depth, but rather acknowledges it as a necessary software layer to run spark on the EC2 infrastructure.

1.3 Hadoop and HDFS

Hadoop is an open source software project that runs on the Java Virtual Machine (JVM). Hadoop was designed to provide similar functionality to Mesos, yet the Spark project is able to use the Hadoop software in conjunction with Mesos. More specifically, we use the Hadoop Distributed File System (HDFS) in our project. HDFS is based off a similar system created by Google called the Google File System (GFS) [3]. One of the main sources of computation time in a large-scale optimization problem would be the serial process of loading in the data set from the hard disk. If the same set of data is going to be analyzed many times on the same cluster, then it would be beneficial to use a distributed system for loading the data. HDFS allows the user to store a set of data as a set of records (which can be thought of as lines in a file). Then, HDFS loads this data on many nodes simultaneously.

For our ADMM algorithm, we must cluster these records in some fashion. The distributed nature of loading by HDFS makes partitioning the data into clusters not a trivial task. We overcome this problem by giving each record an ID. Then clusters are created by doing a "groupBy" call on the modulus of the ID and the number of clusters.

1.4 Work Flow

For each running of the distributed algorithm, there exists a sequence of steps that are reoccurring.

1. Develop new code for testing
2. Compile new code
3. Lease new set of servers
4. Start up servers with Mesos, Spark, HDFS

5. Pull in data set
6. Deploy local optimization code to master
7. Deploy master code to slaves
8. Launch optimization program
9. Save results of program to file
10. Rsync the remote results file to local computer
11. Analyze results in convenient scripting language X
12. Destroy cluster

Steps 3 and 4 were accomplished by heavily modifying a helper script provided by the Spark project. All other steps required many sub-commands and deployment techniques that were hand-crafted by the members of this team.

1.5 Local Deployment Toolset

In order to make such a lengthy process for the workflow manageable, it became necessary to develop a set of high level commands to devise experiments and easily deploy and analyze the results. Firstly, the serialization of statistics from the optimization program was automated via a *StatTracker* class within the scala code we developed, which could then be conveniently serialized to JSON. The prevalence of JSON recently made the reading of this data possible in just about any language. We used python and MATLAB specifically for our analysis. Additionally, the operating system-level commands were wrapped nicely into a set of easy-to-handle python functions, which made experiment deployment for the entire 12 step process above a matter of a handful of lines of python. An example experiment script is given in 1.

Listing 1: Example experiment code in high-level python toolset language developed by team

```
def simple_hdfs_remote_test():
    localfn = '/Users/jdr/Desktop/jazz'
    remotefn = '/root/data'
    launch_cluster(3)
    post_init(False,False)
    store_hdfs('s3.amazonaws.com/admmdata/A.data', 'A.data')
    store_hdfs('s3.amazonaws.com/admmdata/A.data', 'A.data')
    launch_trial(200, nd=nd, nf=nf, fn=localfn)
    run_cmd(rsync_remote(localfn,remotefn, False))
    stop_cluster()
    stats = get_stat(localfn)
```

2 Collaborative Filtering

While ADMM is a more general framework that permits parallelization in numerous ways, we exploit a very particular structure in our problem. The crux of our problem is the large number of samples. For example, in the Reuters corpus data set [5], there are only 40,000 features in the IDF, while there are over 800,000 sample documents. Such a large number of samples makes computation on a single small machine very expensive and difficult. Collaborative filtering [1] gives us a mechanism for splitting the data set across the number of samples.

If we take for example that we penalize each record in an additive manner such that we can express our objective as:

$$\min_x \sum_i f_i(x)$$

where f_i is a function of the estimated feature vector x and the data associated with record i. Then we may equivalently write this problem as:

$$\sum_i \min_{x_i} f_i(x_i) : x_i - z = 0 \forall i$$

Clearly, now the objective function is separable across all samples i, except for the coupling due to the z constraints. The z value acts as *global consensus* variable, since it penalizes local deviations of x_i values from one another. One can see the approach of collaborative filtering as decoupling the sample losses to allow myopic minimization, while penalizing such decouplings via a collecting and comparing step. In the framework of ADMM, such concepts can easily be turned into an iterative algorithm.

In our specific problem, we use collaborative filtering to cluster the document data into a fixed number of batches. Then, these individual batches can be mapped to all the slave nodes discussed in section §1, which greatly reduces the required resources for each individual node. The ADMM collaborative filtering algorithm in addition has the nice property that the master node, which acts as the *fusion center*, does not need any knowledge of data set. Rather, it only acts as an averaging mechanism of the slave nodes. Therefore, the entire data set can be completely distributed, and never needs to be collected at a single node.

3 Reuters Data Set and Sparse Logistic Regression

Lewis et al [5] detail a data set pertaining to Reuters news stories. The data set consists of over 800,000 news stories that were manually classified as belonging to or not belonging to over 100 news topics. The report in [5] goes into grave detail

about the data collection and classification process, which includes a discussion about how humans modified an automated assignment of stories to topics.

The text content of the individual documents was converted into an *Inverse Document Frequency* value for a dictionary of over 40,000 words that existed in the corpus. Such a representation leads to a very sparse A matrix (# samples vs. # dictionary words), since word frequency follows a power law distribution. Additionally, topic membership is also sparse, making our b vector sparse, since most documents would only belong to a couple of topics out of 100.

Implementation Note: Exploiting the sparsity of the data was necessary for the success of our algorithm. For this reason, all data used in this study had a sparse vector representation. This dramatically reduced the data representation size. Additionally, we sought out a java or scala library that would enable sparse matrix calculations. The Colt library had such libraries available, and although the API for its usage was rather verbose, the library was ultimately selected. To be more explicit, we utilized the Parallel Colt library, which builds in parallelization to matrix operation automatically under the hood.

While it is most definitely possible to use all of the topic classifications in a regression analysis to predict the classification of a single topic, we instead focus our problem on determining whether the IDF of a document gives enough information to be able to classify that document into a particular topic. This greatly simplifies our problem.

In the report by Lewis [5], the group analyzes the data set using *Support Vector Machines (SVM),* as a method for regression. We take a slightly different approach in our problem by solving *sparse logistic regression (SLR).* Ignoring the *sparse* modifier, these problems are very similar, as SVM uses a hinge loss function, while SLR uses a logistic loss function. A convenience of our analysis is that the Boyd paper [1] explicitly discusses how one can approach SLR within the ADMM framework.

One key interesting point in our approach is to make our feature solution vector sparse. What this practically means for our specific problem, is that we have a prior belief that only a handful of the 40,000 total words are necessary for predicting whether a document belongs in a topic or not. The output of the SLR solution can then be interpretable by humans. Such benefits of this are using this data for selling ads based on particular news stories, or having users search by relevant words, rather than the actual topic name. A non-sparse solution vector makes displaying the optimization results with some sort of user interface much more difficult and harder to interpret.

4 Shortcomings of Caching in Spark and Proposed Improvements

Currently in Spark, caching is accomplished by explicitly caching a particular RDD, which serves as a starting point for many different transformation processes. While this process fits many use cases, such as Monte Carlo methods for

sampling, our process does not fit into this mold.

In our problem, we have two distinct concept: invariant data, and iterative data. The invariant data for a slice i can be succinctly expressed as C_i where

$$C_i = \left[\begin{array}{cc} -b_i & \operatorname{diag}(b_i) A_i \end{array} \right]$$

This data does not change from iteration to iteration and we would like to cache this data set on each local node. The iterative data is the tuple $\left(x_i^k, u_i^k\right)$,where i is the slice and k is the iteration number. We have the relation for each slice i and each iteration k:

$$\left(x_i^{k+1}, u_i^{k+1}\right) = f\left(x_i^k, u_i^k, z^k, C_i\right)$$

Therefore, the iterative data must be updated with a reference to the invariant data and its previous iteration. If both types of data must never be present outside its local node, then there are currently two options that could be considered. First, there could exist a way of pairing up two *RDD*'s, where one holds the cached invariant data and the other holds a constantly cached iterative data tuple. The functional term for this pairing is a `zip` action. This approach is appealing, as the invariant data would have nice resilience due to its invariance, while the iterative data does not enjoy such reprieve due to its temporal nature. This solution is not ideal as it does a poor job of following the abstraction laid out by the *RDD* object, where absolute location is not of a concern, only relative persistence of location. By zipping two *RDD*'s together, one must view *RDD*'s as a sequence, which is a poor abstraction for distributed data sets.

The second approach would be to directly support a specialized *RDD* type that lets transformation occur on top of a invariant layer. This is the model that was envisioned in our current implementation. A pictorial representation of our approach is given in figure 2. A *DataEnv* is simply an environment which holds functionality related to the invariant data. Let us call this invariant *RDD* an *Environmental RDD*, which gets the point across that this RDD will be used as a environment in which to iterate a sequence of *RDD*'s. Then, subsequent transformations of the *Environmental RDD* will have the environment cached locally, thus saving expense of either serializing the environment locally into cache at every iteration, or even worse, over the wire every iteration. In our example, the subsequent *RDD's* are *LearnEnv*'s, which serve the purpose of learning the optimal feature vector over the *DataEnv*. In our current implementation, we decide that caching locally at every iteration is better than serializing the entire C data set over the wire at every iteration, and this what is implemented in our code. This solution is superior to the `zip` approach, as it does not require the sequential abstraction, and only a single *RDD* needs to be maintained.

While such an implementation has not been worked out at this point, the main mechanism would be allowing particular super-closures of *RDD*'s to be cacheable, which is feature in waiting for Spark.

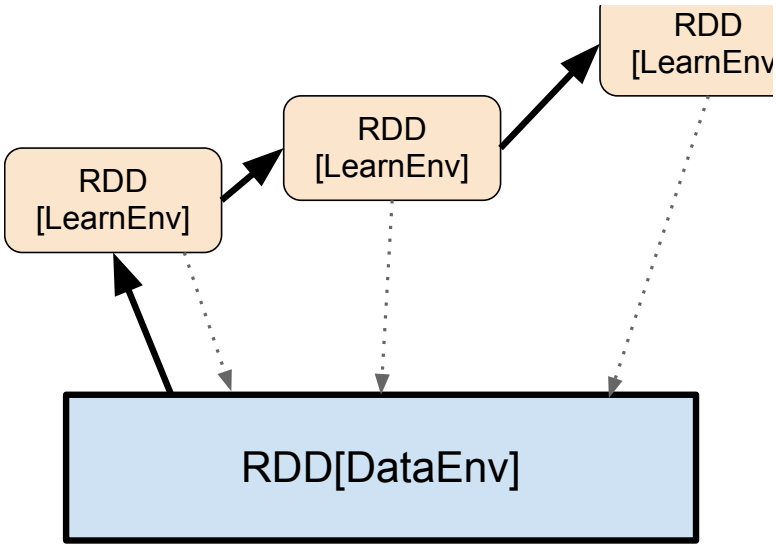

Figure 2: The iterative *LearnEnv* iterates over the invariant *DataEnv*, where both data types are created in a distributed manner.

Foundation of Collaborative Filtering Optimization Framework Not only does this solution work well with our current problem, but we also see it as the basic construct on which all collaborative filtering distributed optimization techniques could be created. Future work is to devise a collaborative filtering library inside of Spark, which allows for optimal slicing of data-sets and automatic scaling of the cluster size to suit the needs of the current data set. At the core of this library will lie the *Environmental RDD* object, which will fully utilize the power of the Spark framework, and enable a style of convex optimization that currently does not exist at super-large scale. Combined with the optimal deployment of slices to number of instances and instance types, this approach can simply add more and more nodes to the current cluster to support more and more data. If particular cost functions are made customizable via some plugin architecture, then a general purpose framework could be developed. For instance, one could allow the customization of desired sparsity along with loss function to give users a large level of customization, permitting *SVM* solutions, *Lasso* solutions, and many more than just *SLR*.

References

[1] Stephen Boyd. Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers. *Foundations and TrendsÂ® in Machine Learning*, 3(1):1–122, 2010.

[2] Jeffrey Dean. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, pages 1–13, 2008.

[3] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. *ACM SIGOPS Operating Systems Review*, 37(5):29, 2003.

[4] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos : A Platform for Fine-Grained Resource Sharing in the Data Center. *Architecture*, 203(UCB/EECS-2010-87):22, 2011.

[5] David D Lewis, Yiming Yang, Tony G Rose, and Fan Li. RCV1: A New Benchmark Collection for Text Categorization Research. *Corpus*, 5:361–397, 2004.

[6] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark : Cluster Computing with Working Sets. *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 39(UCB/EECS-2010-53):10–10, 2010.