Cache-Reduce: Implementing Collaborative Filtering on a Cluster with the Spark Framework

Johann Grauzam Boris Prodhomme Jack Reilly

May 10, 2012

1 Introduction

Our project focuses on using the distributed ADMM algorithm on the Spark framework. The end goal is to compare the running time of the algorithm on both the Spark framework and the traditional Hadoop framework to see the benefits of using in-memory cluster computing techniques. We also discuss implementation details, and other tunable parameters pertaining to the optimization problem.

The ADMM algorithm is described extensively in [3] as a method for distributed optimization. The distributed aspect of computation comes from the separability of the subproblems expressed in the ADMM algorithm. More specifically, for problems such as lasso, support vector machines, and logistic regression, the ADMM can lead directly to a collaborative filtering formulation. For the sparse logistic regression problem, $\min_x \ell\left(Ax-b\right)+\lambda\left\|x\right\|_1$ with logistic loss function $\ell\left(x\right)=\log\left(1+\exp\left(-x\right)\right)$, the A and b can be split across examples to give the parallelized version of the algorithm:

$$\begin{aligned} x_i^{k+1} &= \arg\min_x \left(\ell_i(A_i x) + \frac{\rho}{2}\left\|x - z^k + u_i^k\right\|_2^2\right. \\ z^k &= S_{\lambda/\rho N}\left(\bar{x}^{k+1}+\bar{u}^k\right) \\ u_i^{k+1} &= u_i^k + x_i^{k+1} - z^{k+1} \end{aligned}$$

where all x_i and u_i can be updated in parallel.

This algorithm lends itself particularly well to the map-reduce framework for parallel computing [4], except for one particular downside. Map-reduce is typically involves a single map-reduce process, which means that state need not be persistent over map jobs, since they are one-off jobs. In ADMM, however, this is not the case as this is an iterative algorithm where the A_i, b_i data are needed for each iteration of x_i, u_i. Loading the data for each successive iteration of the algorithm can be expensive, in particular in distributed systems, where network communication is the bottleneck.

The Spark framework was created with these types of tasks in mind. The framework allows one to *cache* variables across map iterations, enabling persistence of states of calculation. We seek to quantify the benefits of such persistence over the standard Hadoop-style map-reduce.

We believe the combination of ADMM parallelization of convex optimization problems and the in-memory distributed programming model of Spark has the potential to be a game-changer for huge-scale model-learning problems. As a proof of concept, we hope to apply this particular strategy to the Reuters Corpus Volume 1 data set (RCV1) [8]. The dataset consists of over 800,000 documents with manually added topic tags. The problem we wish to apply is sparse logistic regression in order understand which keywords are relevant to certain topics. Such problems are very difficult to solve using standard convex optimization algorithms given the size of the dataset (250 GB, 800,000 samples, 50,000 features). We wish to show the performance of our system with that of previous attempts of [8] and standard Hadoop-style map-reduce [4].

The rest of the report is organized as follows. First we present some theoretical background on the ADMM algorithm and Logistic Regression. We then give an overview of the Spark software platform and discuss how it is utilized in our problem. Next we discuss the Reuters data set and how sparse logistic regression can be used to analyze the data set. Then we present the implementation of the ADMM algorithm within the Spark context for specifically a sparse logistic regression problem with sparse data-sets. Then we discuss the details of the distributed architecture and implementation details on the cluster and deployment mechanisms. Our next section gives our main numerical results in the form of a series of experiments that compare results from simulations with varying tuning parameters, architectural parameters, and data set natures. Next we discuss some limitation of the current Spark implementation and a proposal for extending Spark with a data structure more amenable to ADMM applications. Some ideas are given about how our current work could be extended into a general-purpose collaborative-filtering platform. Finally, we discuss future directions.

2 Theory Background

2.1 Dual Ascent

Let consider the following convex optimization problem [2]:

$$\begin{array}{ll} \underset{x}{\text{minimize}} & f(x) \\ \text{subject to} & Ax = b, \end{array} \tag{1}$$

with variables $x \in \mathbb{R}^n$, where $A \in \mathbb{R}^{m\times n}$ and $f : \mathbb{R}^n \to \mathbb{R}$ is convex. The Lagrangian of problem (1) is:

$$L(x, \nu) = f(x) + \nu^T (b - Ax)$$

and the dual function is:

$$g(\nu) = \min_x L(x, \nu) = -f^*(-A^T \nu) - b^T \nu$$

where ν is the dual variable, and f^* is the convex conjugate of f. The dual is given by:

$$\underset{\nu}{\text{maximize}} \qquad g(\nu)$$

with variable $\nu \in \mathbb{R}^m$. Assuming that the strong duality holds, the optimal values of the primal and the dual are the same. Then the following relation holds:

$$x^* \quad = \quad \underset{x}{\text{argmin}}\, L(x, \nu),$$

where x^* and y^* are the two optimal arguments for the primal and the dual problem.

The dual ascent method solves the dual problem using gradient ascent. Assuming that g is differentiable, we can evaluate $\nabla g(\nu)$ with $\nabla g(\nu) = Ax^+ - b$, which is the residual of the equality constraint and where $x^+ = \text{argmin}_x L(x, \nu)$. The dual ascent method is given by this iteration:

$$\begin{aligned} x^{k+1} &:= \underset{x}{\text{argmin}}\, L(x, \nu^k) \\ \nu^{k+1} &:= \nu^k + \alpha^k (Ax^{k+1} - b), \end{aligned}$$

where $\alpha^k > 0$ is a step size, and the superscript is the iteration counter. We notice that this method is called dual ascent because with appropriate choice of α^k, the dual function increases in each step, i.e. $g(\nu^k{+}1) > g(\nu^k)$. The dual ascent can also be used with non differentiable function under some modifications. For an appropriate choice of α^k and under other conditions, we can make x^k and ν^k converge to optimal primal and dual points. However, there is many example where the dual ascent method cannot be used. The dual ascent method is very interesting because it can lead to a decentralized algorithm, meaning that the problem can be split into different peaces, which can be sold in parallel. Then for a separable function f, we get:

$$f(x) = \sum_{i=1}^{N} f_i(x_i)$$

where $x = (x_1, ..., x_N)$ and the variable $x_i \in \mathbb{R}^{n_i}$ are sub-vectors of x. Then we partition the A matrix so that we have:

$$\begin{aligned} A &= [A_1 ... A_N] \\ Ax &= \sum_{i=1}^{N} A_i x_i \end{aligned}$$

Then the Lagrangian is written as:

$$L(x, \nu) = \sum_{i=1}^{N} L_i(x_i, \nu) = \sum_{i=1}^{N} (f_i(x_i) + \nu^T A_i x_i - (1/N)\nu^T b)$$

Then the dual ascent algorithm is given by:

$$\begin{aligned} x_i^{k+1} &:= \underset{x_i}{\text{argmin}}\ L_i(x_i, \nu^k) \\ \nu^{k+1} &:= \nu^k + \alpha^k (Ax^{k+1} - b) \end{aligned}$$

We notice that the x-minimization can be carried out independently, in parallel, for each $i = 1, ..., N$

2.2 Method of Multipliers

The method of multipliers consists in augmenting the convex optimization problem (1). We consider now:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) + (\rho/2)||Ax - b||_2^2 \\ \text{subject to} \quad & Ax = b, \end{aligned} \tag{2}$$

Where $\rho > 0$ is called the penalty parameter. This is an equivalent problem because the equality constraint imposes to have $||Ax - b|| = 0$. Then the augmented Lagrangian becomes:

$$L_{rho}(x, \nu) = f(x) + \nu^T (Ax - b) + (\rho/2)||Ax - b||_2^2$$

The augmented methods are useful to make more robust the dual ascent method for example, and to avoid assumption as strict convexity or finiteness of f. Then applying the dual ascent method to this new problem, we obtain the following algorithm:

$$x^{k+1} := \underset{x}{\text{argmin}}\ L_\rho(x, \nu^k) \tag{3}$$

$$\nu^{k+1} := \nu^k + \rho(Ax^{k+1} - b), \tag{4}$$

This is known as the method of multipliers. This method converges under far more general conditions than dual ascent, including cases when f takes on the value $+\infty$ or is not strictly convex. The improved convergence properties reached thanks to the method of multipliers has a cost. Indeed we notice now that even if f is separable, the augmented Lagrangian L_{rho} is not separable anymore with the basic algorithm given in (3). Then we cannot use decomposition anymore. This is one the reason why the ADMM was created.

2.3 ADMM

The dual ascent method and the method of multipliers are both really useful as described above. ADMM is an algorithm that tries to gather advantages of these two method, in minimizing the cost due to this combination. The problem treated will be under the following shape:

$$\begin{array}{ll} \underset{x,z}{\text{minimize}} & f(x) + g(z) \\ \text{subject to} & Ax + Bz = c, \end{array} \tag{5}$$

with variables $x \in \mathbb{R}^n$ and $z \in \mathbb{R}^m$, where $A \in \mathbb{R}^{p\times n}$, $B \in \mathbb{R}^{p\times m}$, and $c \in \mathbb{R}^p$. The optimal value of the problem (5) is given by:

$$p^\star = \inf f(x) + g(z) | Ax + Bz = c \tag{6}$$

The Lagrangian becomes:

$$L_{rho}(x, y, z) = f(x) + g(z) + y^T(Ax + Bz - c) + (\rho/2)||Ax + Bz - c||_2^2 \tag{7}$$

The ADMM consists in the following iterations:

$$x^{k+1} := \underset{x}{\text{argmin}}\ L_\rho(x, z^k, y^k) \tag{8}$$

$$z^{k+1} := \underset{z}{\text{argmin}}\ L_\rho(x^{k+1}, z, y^k) \tag{9}$$

$$y^{k+1} := y^k + \rho(Ax^{k+1} + Bz^{k+1} - c) \tag{10}$$

where $\rho > 0$. We can see that x and z are updated in an alternating.

2.4 Logistic regression

General optimization problem can be written in the form:

$$\text{minimize} \quad l(Ax - b) + r(x) \tag{11}$$

with $x \in \mathbb{R}^n$, where $A \in \mathbb{R}^{m*n}$ is the weight matrix, m being the number of documents in our problem, and n being the number of features. Also the function $l : \mathbb{R}^m \to \mathbb{R}$ is a convex loss function, and r a convex regularization function. This function let us play on the sparsity of the results.

We suppose that the loss function l is separable and can be written as:

$$l(Ax - b) = \sum_{i=1}^{m} \left\{ l_i(a_i^T x - b_i) \right\} \tag{12}$$

where $l_i : \mathbb{R} \to \mathbb{R}$ is the loss for the i^{th} training set, $a_i \in \mathbb{R}^n$ is the feature vector for the i^{th} example, and b_i the output of the corresponding example. This property is very important for us because we want to create a distributed algorithm. But to be fully distributable, we need to choose a regularization function r also separable, the common choice being $r(x) = \lambda||x||_2$.

The goal of our problem is to classify, either 0 or 1. Then the goal is to find a vector $x \in \mathbb{R}^n$ and an offset $v \in \mathbb{R}$ such that:

$$\text{sign}(a_i^T x + v) = b_i \tag{13}$$

holds for many examples. We call $a_i^T x + v$ a discriminant function. Then the constraint (13) can be written as $\mu_i > 0$ such that $\mu_i = b_i(a_i^T x + v)$; mu_i is called the margin. We generally write the loss functions in function of the margin. Then in our case we have:

$$l_i(\mu_i) = l_i(b_i(a_i^T x + v)) \tag{14}$$

Then the loss functions should be positive and decreasing for negative arguments and zero or small for positive arguments. Indeed, by this, we penalize when there is an error in the classification, that is to say when the margin is negative. Then in order to determine the vector x and v we would like to minimize the following:

$$\frac{1}{m}\sum_{i=1}^{m}\left\{l_i(b_i(a_i^T x + v)) + r(w)\right\} \tag{15}$$

For our problem, the logistic loss function $\log(1+\exp(-\mu_i))$ is well adapted. For the ADMM algorithm, we have the following:

$$\begin{aligned} x_i^{k+1} &:= \underset{x_i}{\operatorname{argmin}}(l_i(A_i x_i) + (\rho/2)||x_i - z^k + u_i^k||_2^2) & (16)\\ z^{k+1} &:= S_{\lambda/(\rho N)}(\overline{x}^{k+1} + \overline{u}^k) & (17)\\ u_i^{k+1} &:= u_i^k + x_i^{k+1} - z^{k+1} & (18)\end{aligned}$$

where $\overline{x}^k$ and $\overline{u}^k$ are averages of `x` and `u` over all the training sets, at time `k`. We notice as well that the `x`-update involve an l_2 regularized logistic regression problem that can be solved by algorithms like L-BFGS. For our project, we decided to use the gradient descent and the accelerated gradient descent methods.

2.5 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 [8], 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 [2] 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 §6, 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.

2.6 Rare events

In our problem, we deal with rare events represented by the positive outputs in the vector b. Indeed, the probability to get a positive value is very small. That means that there is not many positive outputs inside the training sets we use to solve our problem; that explains why we have bad success rates over the positive outputs.

In his paper, King [7] proposed to use a bias correction in parameters estimation but also in the probability estimation of these rare events. This is complicated to realize and we did not have time to take it into account in our project.

3 Reuters Data Set and Sparse Logistic Regression

Lewis et al [8] 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 [8] 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 [8], 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 [2] 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.

Synthetic Dataset In order to test our algorithms on reliable and invariant datasets, we developed code to be able to generate and store randomly created, sparse data sets. This is discussed in more detail in section §6.1.

4 Different software frameworks

4.1 Map-Reduce architecture

MapReduce [4] is a framework for processing highly distributable problems across huge datasets using a large number of computers, collectively referred to as a cluster. In this framework two main steps appear:

- "Map" step: the master computer divides in several sub-problems and send them to the other computers in the cluster. Once these worker computers have solved the sub-problem, they send back the solution to the master computer.
- "Reduce" step: the master computer collects the results of all the sub-problems and combines them in a specific way to obtain a solution to the original main problem.

Provided each mapping operation is independent of the others, all maps can be performed in parallel.

4.2 Hadoop

Hadoop is a free Java framework that supports distributed applications. It enables applications to work with thousands of computational independent computers and petabytes of data. A complete MapReduce algorithm can be used in this framework.

4.3 Spark

Spark [10] is a cluster computing framework built on top of Mesos [6]. As well as Hadoop, it supports distributed applications. The language used is Scala. Spark's programming model is based on two constructs: parallel loops over "distributed datasets", and a limited set of types of shared variables that can be accessed from tasks running on different machines. This is the main advantage compared to Hadoop and the reason why we want to use this framework. Thanks to this architecture, we will be able to compute big calculations only once, and then the results will be shared with all the relevant computers. The framework Spark uses what is called Resilient distributed datasets (RDDs). Their main characteristics are:

- RDDs are composed of immutable data split across the cluster and that can be rebuild if a part of it is lost
- data can be created or modified thanks to "transformation" type methods
- we can also cache data in the cluster, that is to say, we save the result of a calculation. Then, we do not need to recompute each results we already calculated, each time we need them.

Their exists two different types of operations supported by RDDs: the transformation methods and the action methods. TRANSFORMATIONS return a new dataset rather than ACTIONS return a value to the driver program. TRANSFORMATIONS in Spark are lazy. That means that the result is not computed right away. The system will remember the different transformations called and it is only when an action function is called that the calculation are done. Spark is more efficient thanks to this process. When transformations become lazy, then many optimization steps may take place. For instance, if a series of `map` transformations leads to many intermediate calculations that take large amounts of memory, then lazy evaluation can reduce this cost by storing intermediate calculations across all iterates. Such a style of programming suits the *RDD* world well.

5 Algorithm

The ADMM method is a recursive method. Also each iteration of `x`, `z` and `u` are interdependent, that is to say, we need z^k and u^k to compute x^{k+1}, we need all the x^{k+1} and u^k to compute z^{k+1}, and we need x^{k+1}

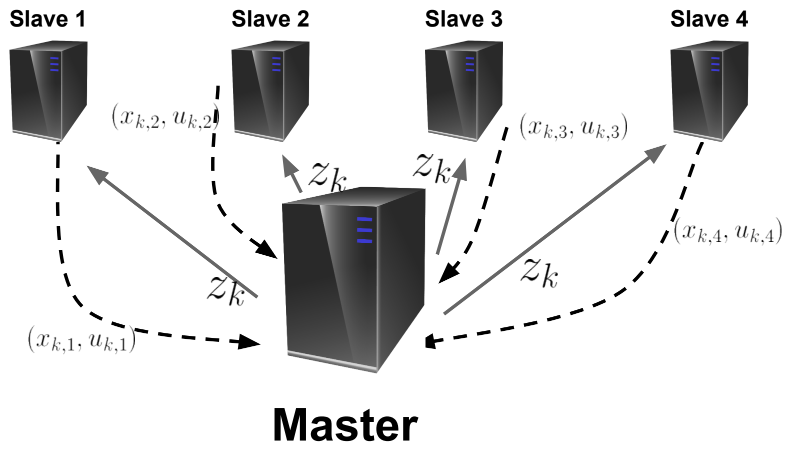


Figure 1: Algorithm fitting cluster's structure

and z^{k+1} to compute u^{k+1}. This makes the code hard because of the nature of the RDDs. Indeed they are composed of immutable variable while we need to store each iteration of `x`, `z` and `u`. In our code, we used the method `cache()` for our RDDs. Indeed, because they gather all our data (samples and outputs we are working on), we did not want to compute again these RDDs each time we call them.

The procedure followed to create these RDDs are a succession of Spark transformations and ends with a `reduce` method. We know that because of the laziness of Spark transformation, values are not computed right away, just the sequence of transformations is save. It is only when we call an action function as `reduce` that the computations are really done. We are using the `cache()` method right after calling reduce.

Figure 1 shows where the different variable are stored into the cluster. So we notice that each slave has a version of `x`, `z`, and `u`. So we understand that we can update `u` on each slave from the `x` and `z` values stored into them. We can do the same when we want to update `x`, by using the `z` and `u` stored in the computer slave. So we understand that the two previous updates can be done on each computer slave. The method to update `z`, however, is different. In the ADMM method, we notice that we need do compute the average of all `x` and all `u` found on each slice. Then we had to use the master computer to update `z`. The method is the following: first the master computer collect all the value of `x` and `u`, then it runs a `z`-update method we created, and finally we broadcasted the updated value of `z`. The broadcast step is necessary to have the same value on `z` on each computer slave. A method already exist on Spark in order to realize this broadcast action. But it was easier for use to create our own method. It would be interesting to use this method in the future.

5.1 xUpdate

The xUpdate equation is given by (19).

$$x_i^{k+1} \quad := \quad \underset{x_i}{\operatorname{argmin}}(l_i(A_i x_i) + (\rho/2)||x_i - z^k + u_i^k||_2^2) \tag{19}$$

As said before, only Spark transformation functions can be used here inside each slave computer in order to update `x`. We notice that the `x` update is obtained by solving an unconstrained optimization problem. To do so, we used two different methods: traditional gradient descent method with backtracking line search algorithm and an accelerated gradient descent method using backtracking as well [1]. A second-order Hessian-based method was attempted for our subproblem, but the cost of computing the Hessian was a limiting factor. A quasi-Newton approach such as L-BFGS [9] would suit this sub-problem well, but time did not permit us to implement this method and compare.

5.2 zUpdate

The zUpdate equation is given by (1).

$$z^{k+1} := S_{\lambda/(\rho N)}(\overline{x}^{k+1} + \overline{u}^{k}) \quad (20)$$

In order to give an example on how the scala code is created, Listing 1 shows how `z` is updated at each iteration:

Listing 1: Z Update Pseudo-code

```
val xLS = oldSet
        .map(.xUpdateEnv)
        .cache()

val z = {
        val reduced = xLS
                .map(ls => ls.x + ls.u)
                .reduce(_+_)

        shrinkage(reduced / nSlices)
}
```

The first step of this code is to create an RDD called `xLS`. As we can see thanks to the `map` method, `xLS` is a RDD composed of elements of type (x^{k+1}, z, u), of all the slices. We have to remember that the `map` method is part of the action functions. It means that this method will "create" a result at the end. It is different from the transformation functions that just modify variables inside an object. Then, the most important part of this code is to cache the RDD we just created. Indeed, if we do not do this, the code will have to compute again the data each time we call `xLS`.

Then comes the zUpdate part. First we create a value "reduced" which consists in copying the x value and then summing with the u value. We do it for all slices as we see with the `map` method. The following `reduce` method sum all these values into one. Then we divide by the number of slices `N` (called here `nSlices`). And then we use the function `shrinkage` that represents $S_{\lambda/(\rho N)}$ in the ADMM algorithm (16). Because we used a `map` method, it will create a value. Then, in order to broadcast these with our original RDD, we had to create a method called `zUpdateEnv`.

In order to update the `z` value, we could have use the accumulator object that exists in Spark. The accumulator would have been useful to realize the summation part over `x` and `u`. Accumulators can only be used through an associative operation but we could not create this type of method for the vectors of the cern library. That is why we did not use the accumulators.

5.3 uUpdate

The uUpdate equation is given by (21).

$$u_i^{k+1} := u_i^k + x_i^{k+1} - z^{k+1} \quad (21)$$

For that part we did not need to create any new variable. We just needed to create a method `uUpdateEnv` that is a transformation function. It will update the current u value using x and z.

5.4 Stopping criteria

The stopping criteria is based on the residuals r^k an s^k defined as:

$$\begin{aligned} r^k &= x^k - z^k \\ s^k &= z^{k+1} - z^k \end{aligned}$$

It has been shown that when these two residuals are small, the objective sub-optimality, $f(x^k) + (z^k) - p^\star$, also must be small. We have th following inequality:

$$f(x^k) + (z^k) - p^\star \leq -(y^k)^T r^k + (x^k - x^\star)^T s^k$$

Because we do not know the optimal value $x^\star$, we cannot use the previous equation as stopping criteria. But we can guess a bond d such that $||x^k - x^\star|| \leq d$. Then we get now:

$$f(x^k) + (z^k) - p^\star \leq ||y^k||_2 ||r^k||_2 + d||s^k||_2$$

Then we can create two parameters ϵ^{pri} and ϵ^{dual} to impose the primal and the dual residuals to be small, i.e.,

$$\begin{aligned} ||r^k||_2 &\leq \epsilon^{pri} \\ ||s^k||_2 &\leq \epsilon^{dual} \end{aligned}$$

where $\epsilon^{pri} \geq 0$ and $\epsilon^{dual} \geq 0$. These tolerances can be chosen such that:

$$\begin{aligned} \epsilon^{pri} &= \sqrt{n+1}\epsilon^{abs} + \epsilon^{rel} max||x^k||_2, ||z^k||_2 \\ \epsilon^{dual} &= \sqrt{n+1}\epsilon^{abs} + \epsilon^{rel}||\rho u^k||_2 \end{aligned}$$

where $\epsilon^{abs} > 0$ is an absolute tolerance and $\epsilon^{rel} > 0$ is a relative tolerance. We have factors $\sqrt{n+1}$ because the l_2 norms are in $\mathbb{R}^{n+1}$. A reasonable value for ϵ^{rel} might be 10^{-3} or 10^{-4}, while the absolute stopping criterion depends on the scale of the typical variable values.

5.5 Map Environment Concept

RDDs are composed of immutable variables. But, because of the nature of the ADMM algorithm, we need to update the value of x, z and u at each iteration. Then, to be able to update these variables, we had to create sub-*RDD*s all pointing to the same variable in a main *RDD*, existing in each slice of our global data set. So, when we update our three variables x, z and u in each slice, we actually create new *RDD*s. This is the point of `RRD[DataEnv]`s which represents the main *RDDs* of each slice, and `RRD[LearnEnv]`s which are the sub-*RDD*s related a `RRD[DataEnv]`s in each slice. It is best to think of these constructs as creating isolated *environments* for the computation of each individual slice to feel as if it had it's own scala environment to live within.

6 Cluster Architecture

To implement the iterative map-reduce [4] 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 2.

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.

6.1 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 .

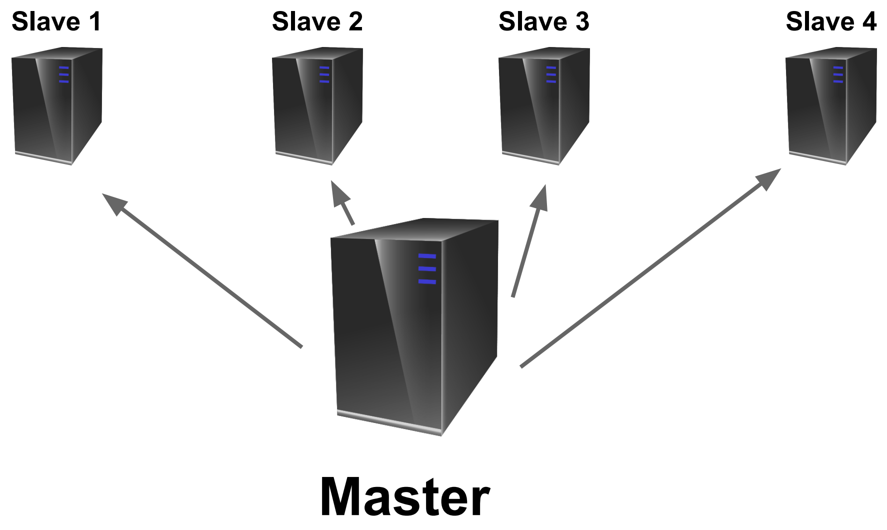


Figure 2: Master/ Slave Architecture

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.

6.2 Mesos Distributed Computing Platform

The Spark [10] software was developed with the Mesos [6] 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.

6.3 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) [5]. 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.

6.4 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

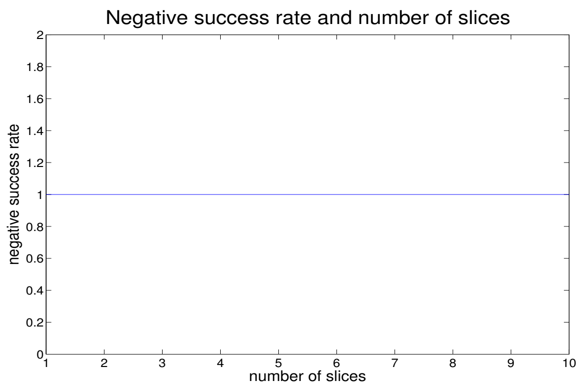
Steps 3 and 4 were accomplished by heavily modifying a helper script provided by the Spark project. All other steps required many sub-commands and deployment techniques that were hand-crafted by the members of this team.

6.5 Local Deployment Toolset

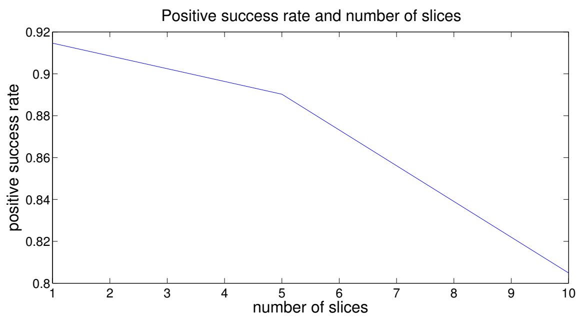
In order to make such a lengthy process for the workflow manageable, it became necessary to develop a set of high level commands to devise experiments and easily deploy and analyze the results. Firstly, the serialization of statistics from the optimization program was automated via a *StatTracker* class 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 Listing 2.

Listing 2: 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)
```



(a) Negative success rate with the number of slices λ



(b) Positive success rate with the number of slices λ

Figure 3: Success rates as a function of slice count.

7 Experiments

7.1 Local experiments

7.1.1 Synthetic data creation

We followed the same procedure as in [2] for generating the synthetic data set. For testing on local machine we generated an instance with $m = 700$ documents and $n = 50$ features. The m documents are distributed among $N = 5$ slices, so each subsystem has 140 training documents. Each feature vector a_i was generated to have approximately 70 nonzero features (a 10% sparsity), each sampled independently from a standard normal distribution. The "true" weight vector $w_{true} \in \mathbb{R}_n$ has 25 nonzero values (a $50\% sparsity$) and these entries, along with the "true" intercept v_{true}, were sampled independently from a standard normal distribution. Then we generate the labels by $b_i = \text{sign}(a_i^T w_{true} + v_{true} + v_i)$ where $v_i \tilde{} \mathcal{N}(0, 0.1)$. We use $\lambda = 0.1$, $\rho = 1$ as default values. As done in [2] we used $\epsilon^{abs} = 10^{-4}$ and $\epsilon^{rel} = 10^{-2}$. Although these last two values should be tuned but a quick analysis showed that the results were not too sensitive to them. It should be noted that the proportion of positive labels randomly generated was around 11%.

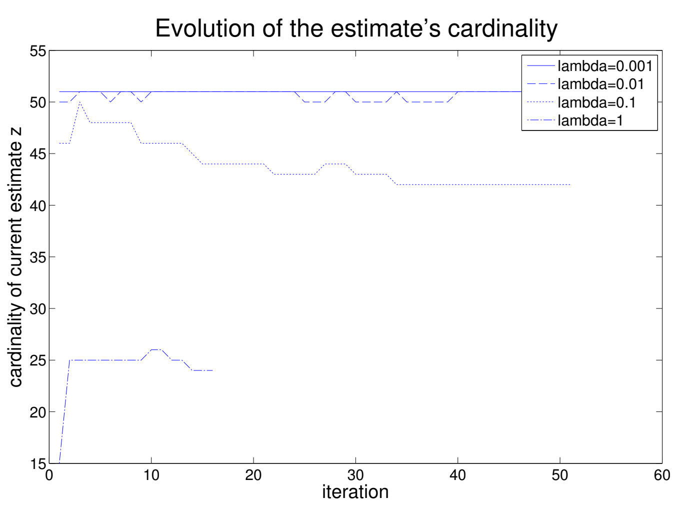
7.1.2 Varying parameters

In this subsection we want to test the following changes in the problem's parameters (while fixing the other ones) :

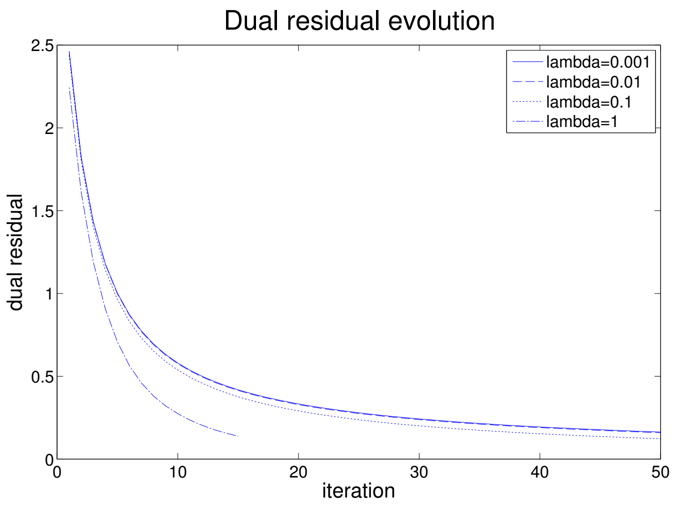
- number of slices : 1, 5 and 10
- λ : $0.001, 0.01, 0.1$ and 1
- ρ : $0.01, 0.1, 1, 10$

The effect of the number of slices We see in Figure 3b that there is a tiny decrease in the success rate when we increase the number of slices but only for the success rate on labels with the smallest proportion (so-called rare events if the proportion is less than 10%).

The effect of the regularization parameter λ It is a known fact that increasing the regularization parameter makes the solution sparser so finding this pattern is reassuring for the validation process. We can actually see that in Figure 4a which shows the evolution of the cardinality of the current estimate for different λ's.



(a) Cardinality for different λ



(b) Cardinality for different λ

Figure 4: Cardinality comparisons as λ varies.

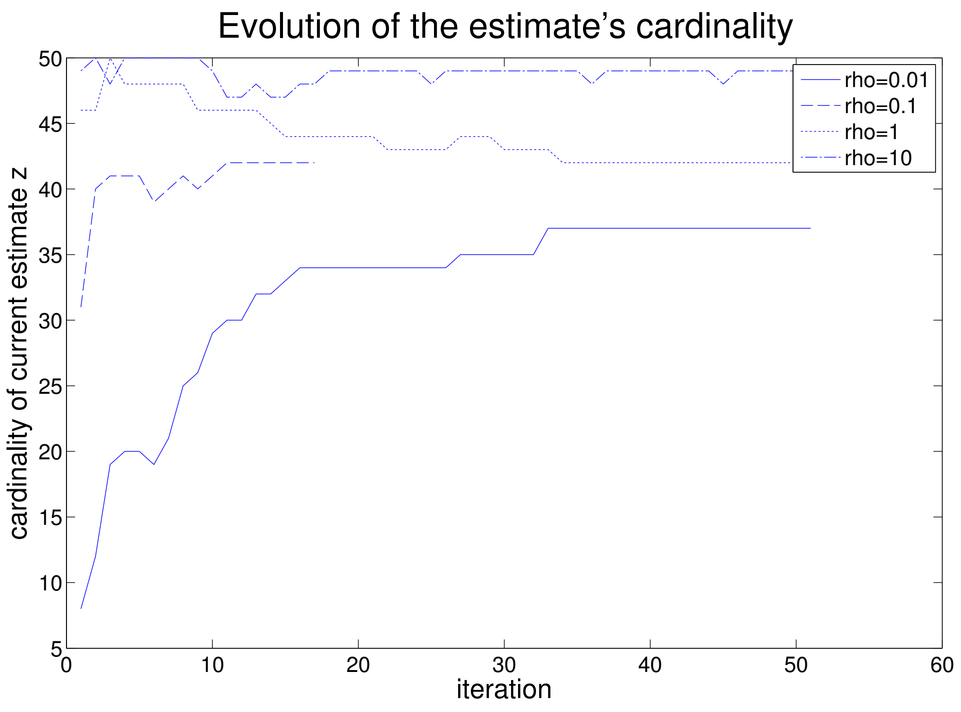
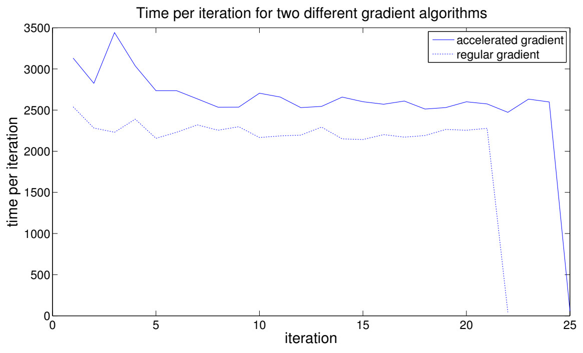
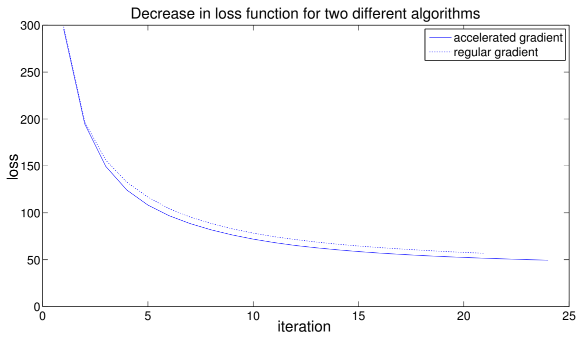


Figure 5: Cardinality for different ρ



(a) Comparison of average time between iterations λ



(b) Decrease in loss for two gradient algorithms λ

Figure 6: Gradient methods comparisons

This parameter λ influences also the number of iterations before meeting the stopping criteria. The Figure 4b shows that it is the dual residual criterion that makes the iterations stop quicker when $\lambda = 1$. In 4b

The effect of the augmented Lagrangian parameter ρ We can see in Figure 5 that similarly to λ, ρ seems to have an influence on the cardinality of the current estimate z^k. ρ influences also the number of iterations as we see that for $\rho = 10$, the number of iterations is less than 20.

7.2 Large-scale experiments

7.2.1 Testing different gradient algorithms

Contrary to what one might think the accelerated gradient does not perform better in terms of average time per iteration. However it seems to perform slightly better in terms of decrease in the loss function. We cannot say much more at this stage of our analysis and one would need to perform a deeper analysis to be able to compare properly the performances of these two gradient algorithms. Since this method only augments the line search value, comparable results are not too shocking.

7.2.2 Comparing different clusters configurations

7.2.3 Experiments on large data sets

To test the EC2 distributed architecture's ability to handle a large dataset, we ran a trial on both the synthetic data and Reuters data with 20,000 documents and 2000 features. The EC2 cluster had 1 master and 13 slaves all running m1.large instance types. The dataset was split into 50 slices.

Below is the performance of the datasets after 20 iterations.

Synthetic data

- proportion of positive events ($b_i = 1$) = 45.3%
- negative success rate = 0.7518
- positive success rate = 0.6104
- total success rate = 0.6909

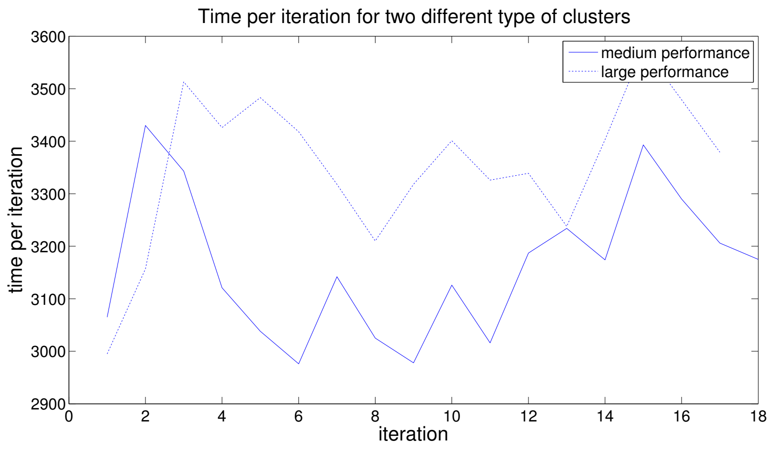


Figure 7: Comparison of average time between iterations λ

Reuters data

- proportion of rare (positive) events : 2.97%
- negative success rate = 1
- positive success rate = 0
- total success rate = 0.9703

While it is disheartening to see the Reuters data-set perform poorly on the rare-event data, we have at least verified the accuracy of the algorithm, as it was able to correctly classify the synthetic data most of the time. We suspect that if more than 20 iterations were run, that the performance would increase on both the synthetic and Reuters data.

Possible explanations for the poor performance on the Reuters set may be the rare event bias, inherent noise in the classification methods of the Reuters humans, and too few iterations.

7.3 K-fold cross validation

Cross-validation is a method used to evaluate how the results of a statistical analysis can be generalized to an independent data set. Actually, we evaluate here its power of prediction. This method uses two kind of data: a training data set where we perform our analysis, and a validation data set used to test the results obtained from the training set. Multiple rounds can be performed, changing the training and the validation sets. Then, to get the final result, we realize a weighted average according to the rate of success of each solution. In the K-fold cross validation, our original set is split into K different sets. We select K-1 sets to create the training set, and the remaining one is used as validation set. Then we process K times such that each sample is used as validation set. Each iteration gives us a vector solution and a success rate. The final solution is obtain in pondering each sub - solution with their success rate.

In the figure 8, we compared the results obtained with the regular ADMM method and the results obtained by using also the K-fold method. The experiment consisted in using 900 documents spilt into 20 slices and considering 100 features. Then over the 20 slices, we considered for each K, K folders on which we applied both the regular ADMM method and the ADMM method with the K fold cross validation method. Thus it gave us two different solutions that we used to calculate the success rate on the $20 - K$ remaining folders.

The results obtained are not really what we were expecting. Indeed could think that the ADMM method combined with the K fold cross validation would have give us better results than when when we use the

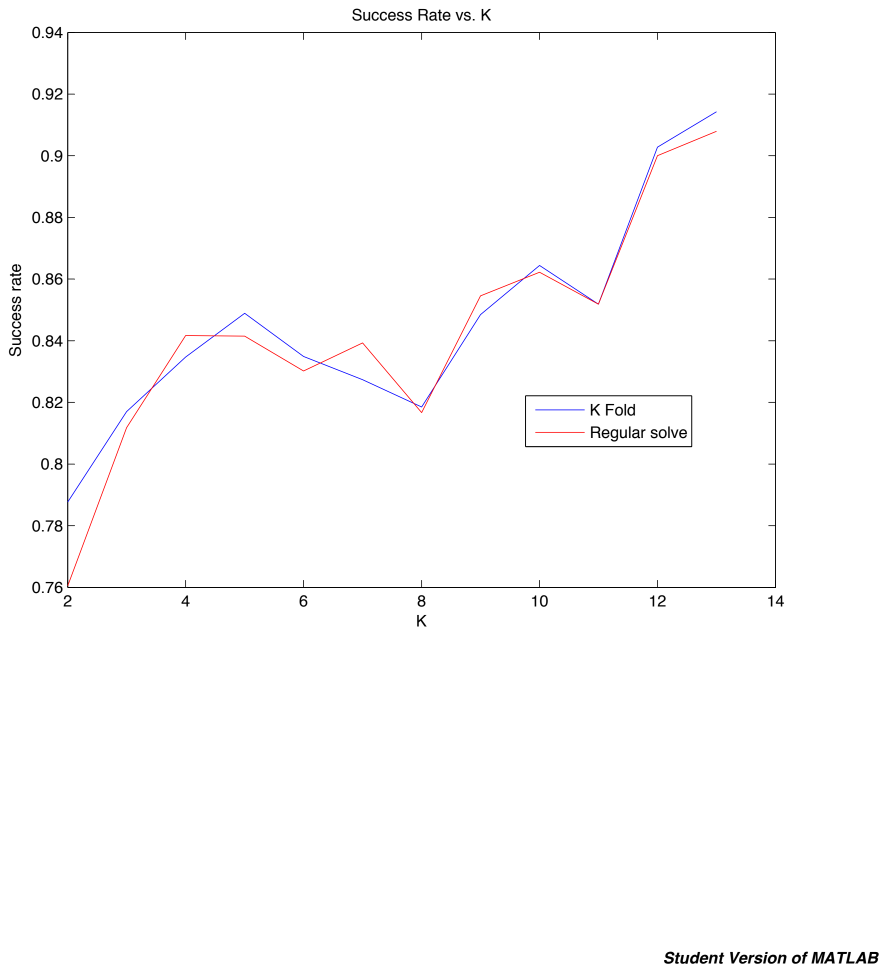


Figure 8: Success Rate vs K

ADMM method alone. Actually the success rates are almost the same in both cases. But we can expect that when we use much more documents and features, we would be able to see a difference between the two methods. But what is very interesting to notice is that when we increase the training set, the success rate on the validation set increases as well. It a result we were expecting.

8 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 9. 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.

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

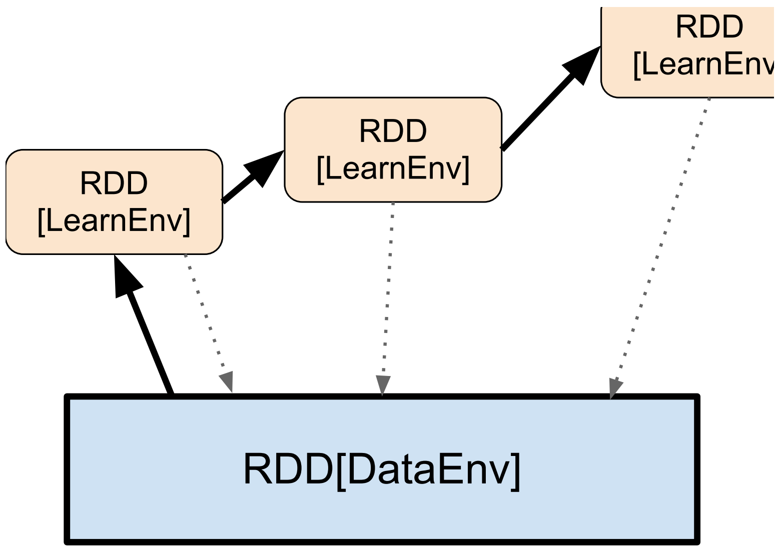


Figure 9: The iterative *LearnEnv* iterates over the invariant *DataEnv*, where both data types are created in a distributed manner.

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*.

9 Future Work

Completion and Analysis of Reuters Data We were not able to complete the analysis of the Reuters Data due to time constraints and some problems with handling rare event data. Once the quota is lifted on the number of EC2 instances allowed at one time and more robust error handling is achieved, then the problem size can easily be scaled up. We will also have to investigate whether the "rare event" phenomenon is actually effecting our results, or if there are other reasons for our poor results on this specific data set at this point.

Splitting across Samples and Features The sparsity in the IDF data set is substantial, due to its power law. This creates the possibility to partition our data along two different dimensions. We have not done research yet into how this can be achieved, but if both dimensions can be reduced in our problem, then additional benefits start to arise, like the ability to use second order methods on the unconstrained minimization sub-problems.

Extending the Framework While we have only discussed the extension of our work to the sparse logistic regression problem, it is evident that our work can be abstracted to a higher level, where we could permit sub-classes to provide the specific cost functions, and our super-class takes care of the work of running Spark and collaborative filtering.

References

[1] Neculai Andrei. An acceleration of gradient descent algorithm with backtracking for unconstrained optimization. *Numerical Algorithms*, 42(1):63–73, 2006.

[2] 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.

[3] L. Boyd, S.P. and Vandenberghe. *Convex Optimization*. Cambridge Univ Pr, 2004.

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

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

[6] 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.

[7] Gary King and Langche Zeng. Logistic Regression in Rare Events Data. *Political Analysis*, 9(2):137–163, 2001.

[8] 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.

[9] Dong C Liu and Jorge Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45(1-3):503–528, 1989.

[10] 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.