2013 IEEE International Conference on Big Data

Practical Distributed Classification using the Alternating Direction Method of Multipliers Algorithm

Peter Lubell-Doughtie and Jon Sondag
Intent Media
New York, USA
{*peter.lubell-doughtie, jon.sondag*}*@intentmedia.com*

***Abstract*—We describe a specific implementation of the Alternating Direction Method of Multipliers (ADMM) algorithm for distributed optimization. This implementation runs logistic regression with L2 regularization over large datasets and does not require a user-tuned learning rate metaparameter or any tools beyond MapReduce. Throughout we emphasize the practical lessons learned while implementing an iterative MapReduce algorithm and the advantages of remaining within the Hadoop ecosystem.**

***Keywords*-distributed algorithms; distributed computing; optimization; predictive models**

I. Introduction

When used at scale, statistical modeling algorithms need to function correctly on large amounts of data in a practical amount of time. In many cases production-scale datasets do not fit into the memory available on a single machine and the dataset needs to be distributed over a cluster.

The MapReduce programming model [1] allows clients to run computations on large distributed datasets with commodity machines. Apache Hadoop is a well known, widely used, and broadly supported MapReduce platform which works with the Hadoop Distributed File System (HDFS) and supports arbitrarily large datasets [2].

The ADMM algorithm [3] is a distributed convex optimization algorithm that can be used to fit machine learning models with convex loss—including linear regression, logistic regression, and support vector machines—and L1 or L2 parameter regularization. We chose to implement ADMM using Hadoop in order to benefit from the wealth of existing Hadoop utilities and the community's extensive knowledge.

Regardless of the problems concerning Hadoop's support for iteration [4], we show that Hadoop can be used to implement an efficient iterative algorithm which is "good enough" in the sense of [5]. That is to say, the benefits of using a reliable platform that easily integrates with our existing toolset outweigh the small increase in runtime per job. To support iteration within Hadoop, we write data from external storage to HDFS, which reduces network latency when transferring large amounts of data. We also use Hadoop's *Input Splits* to persistently partition data between mappers over multiple iterations.

Many challenging problem domains, for example click attribution in online advertising, involve noisy observations of sparse data. Modeling in these domains requires significant amounts of data and an interpretable model. Here we focus on logistic regression with L2 norm.

Traditional batch optimization algorithms for logistic regression fit the model parameters iteratively and require that the training dataset fit into memory for good performance. ADMM also proceeds iteratively but solves an intermediate optimization problem in parallel on subsets of the training data, then combines the intermediate solutions to find a consensus result at each iteration. The time-consuming and memory-intensive intermediate optimization problems are solved in parallel.

When executing long-running modeling jobs hand-tuning parameters can be especially time-consuming. To avoid this we use ADMM with methods that automatically tune parameters while still guaranteeing convergence. This makes for an easy transition from small-scale prototypes fit using batch optimization methods on a sample of the dataset, to large-scale models fit over the entire dataset.

Our main contribution is an adaptation of ADMM that makes large-scale logistic regression user-friendly on a popular platform. Using our implementation, data scientists can accurately fit models without tuning meta-parameters.

The rest of the paper is organized as follows. In section II we formulate logistic regression using the ADMM algorithm and describe the equations necessary to calculate intermediate values. Based upon the formulae and procedures we develop, section III provides a brief review of MapReduce and the necessary details concerning how to implement the ADMM algorithm within Hadoop MapReduce. We present the results of using ADMM to model a production scale dataset of website visitors in section IV. In section V we review recent work related to iterative optimization with Hadoop, and in section VI we conclude.

II. Problem Formulation

In this section we describe exactly how we formulate regularized Logistic Regression with ADMM. Given a matrix of features $\mathbf{A} \in \mathbf{R}^{m \times n}$ and a vector of their labels $b \in \mathbf{R}^m$ with $b_i \in \{-1, 1\}$, our goal is to compute the probability $\Pr(b_i = 1 | \mathbf{A}_i)$ for each row $1 \leq i \leq m$. The first column of the feature matrix represents the intercept and all values in this column are set to 1.

In general, the (not necessarily distributed) convex model fitting problem with regularization can be written as:

$$\text{minimize} \quad \frac{1}{m} \sum_{i=1}^{m} l_i(\mathbf{A}_i^T x - b_i) + r(x), \qquad (1)$$

978-1-4799-1293-3/13/$31.00 ©2013 IEEE

where l_i is the loss for training example i, r is a regularization term, and $x \in \mathbf{R}^n$ is the vector that solves the optimization problem and defines the logistic regression model.

In the case of L2 regularized logistic regression the problem becomes:

$$\text{minimize} \quad \frac{1}{m}\sum_{i=1}^{m}\log(1+\exp(-b_i\mathbf{A}_i^T x)) + \lambda\|x\|_2^2,$$

where λ is the regularization factor.

If we reformulate the above in ADMM global consensus form and reflect the partitioning of training examples across N machines in the cluster, the problem becomes:

$$\begin{aligned}\text{minimize} \quad & \frac{1}{m}\sum_{i_1=1}^{m_1}\log(1+\exp(-b_{i_1}\mathbf{A}_{i_1}^T x_1)) + \ldots \\ & + \frac{1}{m}\sum_{i_N=1}^{m_N}\log(1+\exp(-b_{i_N}\mathbf{A}_{i_N}^T x_N)) + \lambda\|z\|_2^2 \\ \text{subject to} \quad & x_j - z = 0, j = 1, \ldots, N\end{aligned}$$

where $x_j \in \mathbf{R}^n$ is the logistic regression fit on node j. $z \in \mathbf{R}^n$ is a variable representing the global consensus, the vector of parameters that defines the model result. The constraint, $x_j - z = 0$, enforces equality of the parameter estimates across machines.

The ADMM algorithm for the above problem is:

$$\begin{aligned}x_j^{k+1} := \quad & \underset{x_j}{\text{argmin}}\,\frac{1}{m_j}\sum_{i_j=1}^{m_j}\log(1+\exp(-b_{i_j}\mathbf{A}_{i_j}^T x_j)) \\ & + \frac{\rho^k}{2}\|x_j - z^k + u_j^k\|_2^2\end{aligned} \tag{2}$$

$$z^{k+1} := \quad \begin{cases}\bar{x}_q^{k+1} + \bar{u}_q^k & \text{if } q = 1 \\ \dfrac{N\rho^k}{2\lambda + N\rho^k}(\bar{x}_q^{k+1} + \bar{u}_q^k) & \text{otherwise}\end{cases} \tag{3}$$

$$u_j^{k+1} := \quad u_j^k + x_j^{k+1} - z^{k+1}, \tag{4}$$

where k is the iteration number, ρ^k is the penalty parameter (see Section III) for iteration k, and $\bar{x}_q$ represents the average across all mappers of the qth element of the vector $\bar{x} \in \mathbf{R}^n$. Formally, $\bar{x}_q = \frac{1}{N}\sum_{j=1}^{N} x_{q_j}$, and similarly for $\bar{u}$. The first elements of $\bar{x}$ and $\bar{u}$, $\bar{x}_1$ and $\bar{u}_1$, correspond to the intercept and are not regularized, therefore we must treat them differently from the other values in the vector.

$\mathbf{A}_{i_j}$ and b_{i_j} are the portion of the data assigned to and used in mapper j. The values of the x_j are calculated in parallel on the mappers while the values of z and u are calculated on the reducer. The convex minimization problem for each x_j update is solved using the L-BFGS (low-memory BFGS) method [6]. We use a Java implementation of L-BFGS which is self contained and has performed well on Hadoop nodes.[1]

III. Implementation

In this section we describe implementing ADMM for Hadoop MapReduce and provide practical details relevant to building models within a production environment. The MapReduce model divides into three phases: the map, the shuffle, and the reduce. In the map phase the input data is split among a set of mappers and each mapper applies a function to the set of data assigned to it. In the shuffle phase the output of the mappers is assigned to a reducer. In the reduce phase the mapper output is aggregated to create the final values, which are then output by the reducer.

The map and reduce steps can be expressed as:

$$\begin{aligned}\text{map} \quad & (k_1, v_2) \quad \rightarrow \texttt{list}(k_2, v_2) \\ \text{reduce} \quad & (k_2, \texttt{list}(v_2)) \quad \rightarrow \texttt{list}(v_2),\end{aligned}$$

where each mapper has a unique key, k_1. The shuffle is concerned with mapping the intermediate key-value pairs, (k_2, v_2), output by the mappers to the reducers [1].

To implement MapReduce in Hadoop we define classes for the mapper and the reducer, along with a driver that handles input arguments and specifies the mapper and reducer classes (the shuffle is handled internally by Hadoop). In the context of ADMM, each mapper performs the resource intensive task of computing the current x_j. After all mappers have completed their computations, we use a single reducer to compute the z and u_j updates.

A. Persistent Data with Input Splits

When each mapper calculates the x_j^{k+1} values on iteration $k+1$, it must use the u_j^k values that were calculated for the same mapper in the previous iteration. Hadoop does not normally accommodate this sort of persistence. [3] suggests using Apache HBase, a distributed data store, however this would add a new component and its accompanying complexity to the modeling framework.

As an alternative, we use Hadoop's notion of input splits to associate the correct x_j and u_j values with each other, and with the location on HDFS of the data that they correspond to. Input splits specify a *split length* and a *split ID*. These respectively determine the size of the split data and which node to use when performing computations on that data. To ensure that the correct u_j values are loaded when calculating an x_j value, the mapper reads the split ID and chooses the u_i values based on this split ID. The z values are the same in each x_j formula, i.e. across mappers. To reduce transfer cost we replicate the current z values across all mappers at the start of each iteration.

[1] https://code.google.com/p/vladium/

1: **procedure** ADMM($\mathbf{A}$, b, N, $maxIterations$)
2: $k = 0$
3: **while** $notConverged$ and $k < maxIterations$ **do**
4: **for** $j = 1 \rightarrow N$ **do**
5: update x_j^k using Eq. 2
6: **end for**
7: update z^k using Eq. 3
8: **for** $j = 1 \rightarrow N$ **do**
9: update u_j^k using Eq. 4
10: **end for**
11: update ρ^k using Eq. 5
12: $k \leftarrow k + 1$
13: **end while**
14: write z^k to S3.
15: **end procedure**

Figure 1. The ADMM procedure implemented for Hadoop MapReduce. $notConverged$ is a helper function that evaluates the norms to check for convergence.

B. Automatically Updating ρ

We use the reducer to update the penalty parameter ρ. The number of iterations to convergence depends upon ρ, and hand-tuning it can take a significant amount of time because we must often run the algorithm for some time to determine the efficacy of the chosen value.

By varying ρ per iteration we can avoid spending time hand-tuning ρ. We use the scheme suggested in [3] to update ρ on each iteration:

$$\rho^{k+1} := \begin{cases} \tau^{\text{incr}}\rho^k & \text{if } \|r^k\|_2 > \mu\|s^k\|_2 \\ \rho^k/\tau^{\text{decr}} & \text{if } \|s^k\|_2 > \mu\|r^k\|_2 \\ \rho^k & \text{otherwise,} \end{cases} \quad (5)$$

where r^k is the primal residual, s^k is the dual residual, and $\mu > 1$, $\tau^{\text{incr}} > 1$, $\tau^{\text{decr}} > 1$ are user-specified parameters. We have used the recommended values of $\mu = 10$ and $\tau^{\text{incr}} = \tau^{\text{decr}} = 2$.

C. Considerations for MapReduce

Figure 1 shows the coordination algorithm for executing the ADMM iterations. The feature matrix $\mathbf{A}$, the vector of target values b, the number of mapper nodes N, and the maximum number of iterations $maxIterations$, are input to the procedure. The procedure begins by initializing the current iteration number to zero and passing control to the driver. On line 3 the driver checks that the algorithm has not converged and that the number of iterations is less than the maximum number of iterations. We assume there is a helper function $notConverged$, which returns true unless $\|r^k\|_2 \leq \epsilon^{pri}$ and $\|s^k\|_2 \leq \epsilon^{dual}$.

If the driver has not converged and the maximum number of iterations has not passed, we distribute the z and u_i to the mappers and begin the next iteration. Lines 4 through 12 show the x_j^k, z^k, u_j^k, and ρ^k updates that occur in the mappers and reducer on each iteration. Alternatively, if the while loop condition is false, the algorithm is complete and we output the current vector of predicted weights, z^k, to persistent storage and exit, as show on line 14.

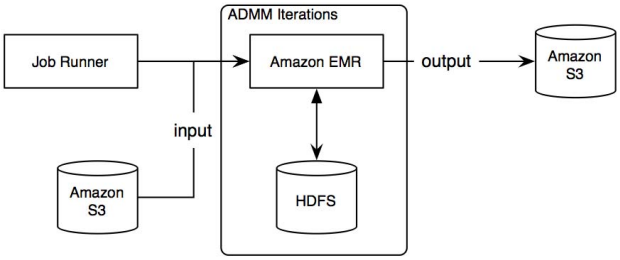

Figure 2. The Job Runner sets paths for the input data, the output data, the Jar file with ADMM, and the value of any adjustable parameters, e.g. the number of iterations. ADMM computes and stores intermediate data in HDFS, then after the final iteration, the problem solution (z^k in ADMM notation) is output to S3.

The output data is written to, and the input data is read from, Amazon's simple storage service (S3). To reduce the network latency involved in transferring data from S3 to the MapReduce nodes, we store the data—feature matrix $\mathbf{A}$ and target values b—directly on the nodes via HDFS. An advantage of running Hadoop on Amazon's Elastic Map Reduce (EMR) service and using data stored in S3 is that there are no transfer fees when moving data between these two systems. The infrastructure workflow is shown in Figure 2.

Wherever feasible, we have made our implementation of the ADMM algorithm generic. Arguments to the driver allow users to exclude columns from the input data, optionally add an intercept, set the initial ρ, set the maximum number of iterations, and set the input and output locations. We have released an implementation of ADMM for Hadoop as an open source Java library.[2]

IV. Results

As part of our modeling pipeline, we run the ADMM algorithm daily on rolling historical data consisting of approximately 25 million records with 440 features per row. Our training examples are real vectors representing website visit attributes and our labels are real scalars representing each visits known value. The goal is to build a model that accurately segments unseen visitors by predicted value based upon their attributes.

As depicted in Figure 2, we load the data and a Jar file containing the ADMM algorithm from S3 into EMR. To evaluate the performance of our implementation of the ADMM algorithm, we examine the output of one day's run. We measure performance by comparing the change in loss function per iteration. We have also examined equivalent

[2]https://github.com/intentmedia/admm

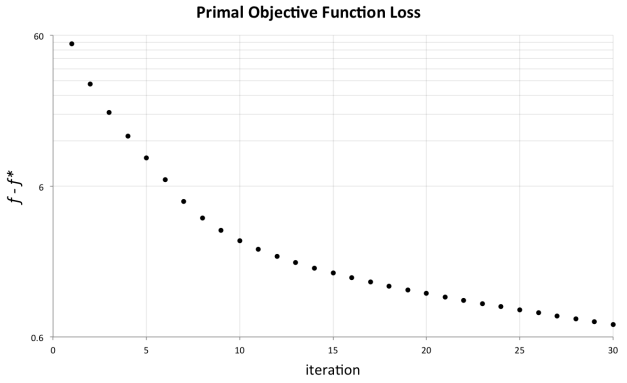

Figure 3. The difference between the model's predictions of the primal objective values and the "true" value as approximated by the model after many iterations. The loss decreases dramatically at first and then continues to decrease during later iterations.

results from other days and verified that the results do not vary substantially from day to day.

We plot the progress of our algorithm by showing that the value we are minimizing in Equation 1 decreases per iteration. Figure 3 shows this through the estimated loss in primal objective value, which we calculate by subtracting from the current primal objective value an approximated minimum primal objective value taken after many iterations (75 in this case). The estimated loss is plotted for each of the first 30 iterations on a logarithmic scale. We see that the loss decreases, and therefore that the accuracy consistently improves as more iterations pass. On a cluster of 20 high memory quadruple extra large EMR instances, 25 iterations run in 1 hour and 25 minutes.

V. Related Work

The algorithm and our implementation are based upon [3], which introduced the ADMM algorithm. [4] presents a more efficient approach to iterative computation in Hadoop requiring modifications to core Hadoop code, which reduce the time needed to transfer data and initialize MapReduce jobs. This is not the approach we took as we remained within the Hadoop ecosystem and avoided the ramp-up costs associated with a less well known solution. Based on our experience, MapReduce is "good enough" for ADMM and logistic regression. We evaluate the cost as described in [5], and conclude that the reduction in implementation costs gained by using Hadoop outweigh the performance improvements that could be gained by using a less well-known platform.

[7] presents a MapReduce implementation of tree ensemble learning and uses a wrapper for MapReduce to coordinate iterations of the underlying algorithm. [8] presents a distributed method for classification that is similar to random forests. Apache Mahout contains an implementation of logistic regression that uses stochastic gradient descent (SGD).[3] As noted in [9], this SGD implementation is sequential, which substantially limits the performance of the algorithm and makes it difficult to scale. To address this, [9] presents a scalable convex loss system that uses a Hadoop compatible variant of MapReduce.

VI. Conclusion

Using an ADMM implementation for MapReduce allows a simpler workflow when moving modeling ideas from prototype to production-scale. We showed a specific example of using the ADMM algorithm to build a logistic regression model. Importantly, ADMM is a generic optimization algorithm that can be applied to other problems with minimal modification. Our contribution is a distributed implementation on Hadoop that allows users to experiment with many different models at scale. In future work we plan to support additional convex loss optimization problems.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10.

[2] T. White, *Hadoop: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2009.

[3] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Found. Trends Mach. Learn.*, vol. 3, no. 1, pp. 1–122, Jan. 2011.

[4] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: efficient iterative data processing on large clusters," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 285–296, Sep. 2010.

[5] J. Lin, "Mapreduce is good enough? if all you have is a hammer, throw away everything that's not a nail!" *CoRR*, vol. abs/1209.2191, 2012.

[6] J. Bonnans, *Numerical optimization: theoretical and practical aspects*, ser. Universitext (1979). Springer-Verlag New York Incorporated, 2003.

[7] B. Panda, J. S. Herbach, S. Basu, and R. J. Bayardo, "Planet: Massively parallel learning of tree ensembles with mapreduce," in *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB-2009)*, 2009.

[8] J. Lin and A. Kolcz, "Large-scale machine learning at twitter," *SIGMOD*, 2012.

[9] A. Agarwal, O. Chapelle, M. Dudík, and J. Langford, "A reliable effective terascale linear learning system," *CoRR*, vol. abs/1110.4198, 2011.

[3] https://cwiki.apache.org/confluence/display/MAHOUT/Logistic+Regression