Alternating Direction Method of Multipliers Implementation Using Apache Spark

Dieterich Lawson

September 25, 2014

1 Introduction

Many application areas in optimization have benefited from recent trends towards massive datasets. Financial optimization problems ingest decades of fine-grained stock history and recent energy grid optimization techniques optimize hundreds of millions of variables associated with hundreds of thousands of devices [KCLB13]. These massive datasets improve results but in turn require algorithms that work at massive scale. We consider a distributed method for solving large-scale optimization problems called Alternating Direction Method of Multipliers (ADMM) and have created an open source ADMM implementation using Apache Spark.

2 Algorithm

A popular approach to distributed optimization is the Alternating Direction Method of Multipliers , which assumes that the objective of an optimization problem can be decomposed into a sum of subproblems. Generally, each subproblem is first solved separately, and then all subproblem solutions are combined to form a global solution. Put another way, the second step enforces a constraint that all subproblems have consistent solutions. Formally, ADMM solves problems of the form

$$\begin{aligned} \text{minimize} \quad & f(x) + g(z) \\ \text{subject to} \quad & Ax + Bz = c \end{aligned}$$

with $x \in \mathbf{R}^n$, $z \in \mathbf{R}^m$, $A \in \mathbf{R}^{p \times n}$, $B \in \mathbf{R}^{p \times m}$, and $c \in \mathbf{R}^p$. We also assume that f and g are convex with codomain $\mathbf{R} \cup \{+\infty\}$. Because f and g can take on extended values, they can be used to encode constraints by taking on $+\infty$ when a constraint is violated and 0 otherwise.

In our specific problem formulation, we take $A = I$, $B = -I$, $c = 0$, and assume that f decomposes into a sum of functions. This yields the problem

$$\begin{array}{ll} \text{minimize} & \sum_{i=1}^{N} f_i(x_i) + g(z) \\ \text{subject to} & x_i - z = 0, \quad i = 1, \ldots, N \end{array} \tag{1}$$

where $x_i \in \mathbf{R}^n$ are variables local to each subproblem and $z \in \mathbf{R}^n$ is a 'consensus variable' that ensures that all x_i are equal. This form of ADMM is called the consensus form with regularization because consensus is forced between the x_i's, and g often represents a regularization function. We chose to implement this form of ADMM in our library because it is reasonably general and many of the most common ADMM problems can be represented in this form.

ADMM solves this optimization problem by performing gradient ascent on the augmented Lagrangian of (1), which is discussed at length in [BPC+11]. Concretely, the algorithm takes the form

$$\begin{aligned} x_i^{(k+1)} &:= \underset{x_i}{\operatorname{argmin}} \left(f_i(x_i) + (\rho/2)\|x_i - z^{(k)} + u_i^{(k)}\|_2^2 \right) \\ z^{(k+1)} &:= \underset{z}{\operatorname{argmin}} \left(g(z) + (N\rho/2)\|z - \bar{x}^{(k+1)} - \bar{u}^{(k)}\|_2^2 \right) \\ u_i^{(k+1)} &:= u_i^{(k)} + x_i^{(k+1)} - z^{(k+1)} \end{aligned} \tag{2}$$

where the superscripts denote iteration number and ρ is a parameter that mediates the tradeoff between local optimality and global consensus.

The iteration for x_i and z, (2) and (3), require computing the proximal operator of f_i and g. In fact, the ability to evaluate f and g is not necessary as long as the proximal operators of f_i and g are provided. This characteristic of ADMM allows our library to be general and extensible – any function for which the proximal operator is known can be used. Furthermore, many functions have closed form or easily computable proximal operators so computing the proximal operators is often fast and simple.

3 Apache Spark

Apache Spark is a framework that lets users perform in-memory computations on clusters in a fault tolerant manner [ZCD+12]. Specifically, it allows a user to load files from a distributed filesystem into memory across a cluster and perform computations on those files while abstracting away the difficult parts of distributed computation such as fault-tolerance and data distribution.

Spark was created specifically to improve the performance of iterative algorithms over similar frameworks like Hadoop [ZCD+12]. Spark differentiates itself by supporting iterative algorithms out of the box and keeping data in memory between iterations, neither of which Hadoop or similar frameworks do. These features are particularly useful when implementing ADMM because it is an inherently iterative algorithm and the f_i are often the same function computed across different subsets of data.

For example, computing an iteration of ADMM inevitably requires multiplying matrices, but in Hadoop those matrices are read from disk into memory, multiplied, and then written back out to disk every iteration. With Apache Spark, the data used for each iteration can be cached in memory so that the matrices don't have to be read from disk every time, yielding a dramatic speedup. Furthermore, Spark handles iterative computations natively but Hadoop has no such support.

Spark also differentiates itself from GPU-centric solutions by focusing on 'commodity computing', i.e. using widely available server hardware that may not have specialized GPUs. This limits the speed of solutions built on Spark, but allows it to scale using any available servers. It also makes Spark more accessible to users outside the numerical computing community, as commodity clusters are easily accessible through services like Amazon's AWS.

4 Implementation

In the Spark framework, the iteration (2) takes the form of Algorithm 1. Note that the u_i and x_i updates in steps 3 and 4 are carried out in parallel for each subproblem and so are distributed across the workers in the cluster. The averaging of u_i and x_i (steps 5 and 6) require a distributed sum, the details of which Spark handles. After the computation of z in step 7, the result is sent to all workers in the cluster via a mechanism called 'broadcast' that Spark provides. Finally note that in our implementation, subproblems need not correspond exactly to workers – one worker can handle many subproblems.

Algorithm 1 Consensus ADMM on Apache Spark

1: **initialize** N subproblems, each with their own x_i, u_i, and copy of z
2: **repeat**
3: $\quad u_i := u_i + x_i - z$
4: $\quad x_i := \operatorname{argmin}_x(f_i(x) + (\rho/2)\|x - z + u_i\|_2^2)$
5: $\quad \overline{x} := \frac{1}{N}\sum_{i=1}^N x_i$
6: $\quad \overline{u} := \frac{1}{N}\sum_{i=1}^N u_i$
7: $\quad z = \operatorname{prox}_{g,N\rho}((\overline{x} + \overline{u})/N)$
8: $\quad$ **broadcast** z to N subproblems
9: **until** convergence

4.1 Code examples

As stated in §2, our library solves optimization problems written in the consensus form with regularization, (1). Users need only pass their selections of f and g to our solver. Consider a standard Lasso problem:

$$\text{minimize} \quad \|Ax - b\|_2^2 + \lambda\|x\|_1$$

where $A \in \mathbf{R}^{m \times n}$, $b \in \mathbf{R}^m$, $x \in \mathbf{R}^n$ is the optimization variable, and λ is a weight that controls the amount of L^1 regularization. The following code solves the Lasso problem using our library.

```
val A = new BlockMatrix(sc.textFile(A_file), blockSize)
val f = L2NormSquared.fromMatrix(A, rho)
val g = new L1Norm(lambda)
var admm = new ConsensusADMMSolver(f, g, abstol, reltol, sc)
admm.solve(rho, maxiters)
```

The first line loads the matrix, A, from the distributed file system into memory across the cluster, while the second line uses A to create the function f as $\|Ax - b\|_2^2$. The vector b is implicitly specified as the last column of A. The third line selects $\lambda\|x\|_1$ as g, and the fourth line creates an instance of the solver and sets the numerical tolerance parameters. The final line solves the problem using step size ρ.

Training a support vector machine (SVM) is similar.

```
val A = new BlockMatrix(sc.textFile(A_file), blockSize)
val svm = SVM(A, rho, C, 'radial')
var admm = new ConsensusADMMSolver(svm.f, svm.g, abstol, reltol, sc)
admm.solve(rho, maxiters)
```

In this case A is a matrix of training examples and classes, and the user is also asked to specify the parameter C and the type of kernel for the SVM.

The speed of convergence of many ADMM problems is heavily dependent on the choice of ρ, so we allow users to provide a function that computes ρ each iteration. Here is a quadratic program that uses an absolutely summable ρ where the solution x, is constrained as $0 \leq x \leq 1$.

```
val as_rho = (iter: Int)  => { 1/(iter.toDouble*iter) }
val f = L2NormSquared.fromMatrix(A, ss_rho)
val g = new GeqConstraint(0.0, 1.0)
var admm = new ConsensusADMMSolver(f, g, abstol, reltol, sc)
admm.solve(as_rho, maxiters)
```

4.2 Function library

To date we have implemented a small set of functions with proximal operators that can be used to specify f and g. The functions that can be used as either f or g are:

- $f_{A,b}(x) = \|Ax - b\|_2^2$
- $f_A(x) = \mathbf{1}^T(Ax + \mathbf{1})_+$
- $f_{\lambda_1,\lambda_2}(x) = \lambda_1\|x\|_1 + \lambda_2\|x\|_2^2$

- $f(x) = \text{huber}(x)$

Note that the second function, when given an A properly constructed from training examples and class labels, is hinge loss on the misclassification error, i.e. a support vector machine. For specific details on how such an A is constructed see [BPC+11]. Additionally, the third function is commonly known as elastic net regularization, and subsumes both the L^1 and L^2 regularization cases when $\lambda_2 = 0$ and $\lambda_1 = 0$, respectively.

In addition to the functions above, g can also be chosen as $g_{a,b}(x) = I[a \leq x \leq b]$, where I denotes an indicator function that takes on ∞ when the condition is violated and is 0 otherwise. This function can only be chosen for g because it encodes a constraint, and so to guarantee that the solution, z, is feasible it must be applied through g.

5 Numerical results

We tested our solver on a Spark cluster that we created using Amazon AWS. We used 8 'm3.xlarge' machines as workers with 15 Gb of memory each, and randomly generated a series of dense lasso problems. In each problem A had 500 columns and we varied the number of rows. We used the same set of parameters for all solves. Information for each solve can be seen in Table 1. Note that the total time to solve is roughly linear in the number of rows.

6 Conclusions and future work

We have created a solver for optimization problems expressed in consensus form using Apache Spark. Currently, we are working with members of the Spark community to include our library in a future release of Spark. Until that time, our library will be available at

`https://github.com/dieterichlawson/admm`

In the future we also hope to make performance improvements to our library, as well as improving the selection of functions and constraints.

Table 1: Timings for several dense lasso solves

Number of rows	10^3	10^4	10^5	10^6	10^7
Nonzero entries	5×10^5	5×10^6	5×10^7	5×10^8	5×10^9
Total time (s)	21.6	44.7	65.4	89.2	520.1
Number of iterations	349	234	128	81	15
Average iteration time (s)	0.062	0.191	0.511	1.101	34.673

References

[BPC+11] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine Learning*, 3(1):1–122, 2011.

[KCLB13] Matt Kraning, Eric Chu, Javad Lavaei, and Stephen Boyd. Dynamic network energy management via proximal message passing. *Foundations and Trends in Optimization*, 1(2):1–54, 2013.

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.