ADMM Implementation on Apache Spark

Dieterich Lawson
EE364b: Convex Optimization II Class Project

Introduction
Recent trends towards massive datasets have spurred the creation of cluster computing frameworks such as Hadoop and Spark that can utilize commodity hardware instead of specialized GPUs. Although these frameworks are rising in popularity, no ADMM solver yet existed for them that was open source and sufficiently general. We implemented such a library on top of Spark, which is a framework built to support iterative algorithms on massive datasets.

Consensus ADMM
Our library solves ADMM problems expressed in the consensus form with regularization, which can be written as

\[
\min_{x \in \mathbb{R}^N} \sum_{i=1}^{N} f_i(x_i) + g(z) \\
\text{subject to } x_i - z = 0, \quad i = 1, \ldots, N
\]

where \( x_i \in \mathbb{R}^d \) and \( z \in \mathbb{R}^d \). The functions \( f_i \) and \( g \) are extended-real valued, and \( g \) often represents a regularization function. The consensus form was selected because many common problems such as lasso, distributed support vector machines (SVM), logistic regression, and quadratic programs can be written in this form.

Algorithm
The consensus form ADMM problem can be solved using the following iteration:

\[
x_i^{k+1} := \arg\min_{x_i} \left( f_i(x_i) + \rho/2 \| x_i - z^{(k)} + u_i^{(k)} \|^2 \right)
\]

\[
z^{k+1} := \arg\min_{z} \left( g(z) + \rho/2 \| z - x^{(k+1)} - u^{(k)} \|^2 \right)
\]

\[
u_i^{k+1} := u_i^{(k)} + x_i^{(k+1)} - z^{(k+1)}
\]

Specifically, our library accepts the functions \( f_i \) and \( g \) and requires that they each define a prox operator. It then uses the following algorithm.

Algorithm: Consensus ADMM on Apache Spark
1. Initialize \( N \) subproblems, each with their own \( x_i, u_i, \) and copy of \( z \)
2. Repeat
3. \( u_i := x_i - z \)
4. \( x_i := \arg\min_{x_i} \left( f_i(x_i) + \rho/2 \| x_i - z + u_i \|^2 \right) \) (the next two steps require a distributed sum across \( N \) subproblems)
5. \( z = \frac{1}{N} \sum_{i=1}^{N} x_i \)
6. \( \tau = \frac{1}{N} \sum_{i=1}^{N} u_i \)
7. \( z = \text{prox}_{\rho g}(z - \frac{1}{N} \sum_{i=1}^{N} u_i) \)
8. Broadcast \( z \) to \( N \) subproblems
9. Until convergence

In our implementation, 'subproblems' do not correspond exactly to workers—on worker can handle many subproblems.

Code examples
The following code solves a dense lasso system.

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

Training a support vector machine is similar.

```scala
val svm = SVM(A, rho, C, 'radial')
val g = new GeqConstraint(0.5)
val admm = new ConsensusADMMSolver(svm.f, svm.g, abstol, reltol, sc, scratch_dir)
admm.solve(as_rho, maxiters)
```

Function library
Our library has a small but growing selection of functions that can be used for \( f \) and \( g \). We have completed or are working on the following functions that can be used for \( f \) or \( g \):

- \( f_A(x) = \| Ax - b \|^2 \)
- \( f_A(x) = 1^T (Ax + 1) \) (SVM)
- \( f_{\lambda_1, \lambda_2}(a) = \lambda_1 \| a \|^2 + \lambda_2 \| x \|^2 \) (elastic net)
- \( \text{huber}(x) \)

And this function, which encodes constraints, can be used for \( g \):

- \( g_{a,b}(x) = I(a \leq x \leq b) \)

Additionally, our library was constructed to be very extensible so it is easy to add new functions.

Numerical examples
We tested our solver on a Spark cluster that we created on Amazon EC2. We used 5’m3.xlarge’ machines as workers with 15 Gb of memory each, and randomly generated a dense lasso problem with \( A \in \mathbb{R}^{500 \times 10^4} \). On disk, \( A \) was roughly 37 Gb.

Results
At the beginning of the solve Spark must load \( A \) into memory, which took roughly 5 minutes. The solve itself took 13 minutes and 36 seconds at an average of 2.2 seconds per iteration. Note that this is a log-log plot, and the algorithm converged to usable precision in roughly 10 iterations, or 20 seconds.

We also tried a smaller lasso problem, with \( A \in \mathbb{R}^{10^5 \times 500} \). In this case it took roughly 3 minutes to load \( A \) into memory and 144 iterations to converge. The solve took and 3 minutes and 7 seconds at an average of 1.3 seconds per iteration, and converged to usable precision (\( 10^{-5} \)) within 6 iterations or 8 seconds.

Future work
We are currently working to submit this solver for inclusion in the Apache Spark project, and hope to continue that effort. We also plan to add more functions to the library and optimize the solver for greater speed. We would also like to add support for solving multiple optimization problems at the same time, which is useful in fitting regularization paths.

Acknowledgements
We would like to thank Reza Zadeh, Madeleine Udell, and Ernest Ryu for suggesting this project and for their guidance. This work is heavily based on papers written by Neal Parikh and Stephen Boyd.