CME 323 – Distributed Algorithms and Optimization
Final Project Report

Alternative Least Squares in Spark

Patrick Landreman

Introduction

In an upcoming research project, I will be evaluating Collaborative Filtering (CF) methods for the prediction of patient medical data. CF refers to an approach for estimating unknown entries in a matrix using the known entries. There are a host of algorithms which produce this estimate by factoring the matrix into a product of two matrices. Apache Spark includes an algorithm known as Alternating Least Squares (ALS).

To understand the core idea of ALS, we will define our incomplete matrix of data as *M.* CF, in general, seeks to express this matrix as the product

$$PQ = \hat{M} \qquad \text{(Eq. 1)}$$

where $\hat{M}$ then contains estimates for the unknown values of *M.* The core concept of ALS is to fix the entries of either *P* or *Q* and solve for the other, noting that the solution is that of a convex least-squares optimization problem. For example, the solution for the *j*th column of Q is given by

$$q_j = \left(P^T S P + \lambda I\right)^{-1} P m_j \qquad \text{(Eq. 2)}$$

where

$$S_{ii} = \begin{cases} 1 \text{ if } M_{ij} \text{ is known} \\ 0 \text{ otherwise} \end{cases}$$

If the matrix *M* has dimensions *n* x *m*, then P and Q will have dimensions *n x k* and *k x m* respectively. *k* is a hyperparameter chosen to estimate the number of "latent factors"; that is, the rank of *M*.

For my final project, my objective was to deepen my understanding of CF and the Spark framework by attempting my own implementation of ALS. I would then compare the performance of my own algorithm against the provided one to gain insight into the nuances of the implementation. In this report, I will provide an overview of how my algorithm is designed and executed in Spark/Scala, as well as example performance data when applied to the MovieLens user-movie ratings dataset.

Implementation

In using Spark, it is assumed that the size of M is prohibitive to store on a single computer. In my algorithm, which will be called *pALS*, I assume that k is sufficiently small that both P and Q fit on a single machine. In this case, it is possible to perform iterative solutions by broadcasting whichever matrix is currently fixed to all worker machines. Each worker receives the entries of M corresponding to a single column and solves for a single column $q_{j.}$ Equivalently, when solving for P one can first take the transpose of Eq. 1, and then each worker receives a single row of M and solves for p_i. The matrices S and λI are never explicitly created – rather, the computation is performed by iterating over the diagonal elements of P^TSP.

pALS is compared against Spark's ALS implementation (*sALS*) using the MovieLens dataset. The dataset is split randomly into training, validation, and test sets. Each model is trained using several combinations of rank, regularization parameter, and number of iterations. After training, the Root Mean Squared Error (RMSE) is computed for the validation test set. The tests were performed locally on my MacBook Pro.

Results

By the end of the project, I was able to bring *pALS* to a point where the code executes to completion. It is unclear that the algorithm is producing the desired model, as the RMSE remains constant for all hyperparameter combinations, whereas the RMSE for *sALS* changes considerably. Notably, *pALS* runs dramatically slower (see Table 1 for trial data). There are several causes for this, as outlined in the following.

pALS should not be expected to run as quickly because of the distribution of the factor matrices P and Q. In particular, in *pALS* these matrices must be communicated across the network (one-to-all) each iteration. It is not necessary for the entire factor matrix to be present on a worker machine; those elements which are mapped to zero by multiplication with S are never used. *sALS* takes advantage of this and thus sends smaller amounts of data over the network.

The performance of *pALS* is limited further due to suboptimal implementation of matrix operations. I decided it would be simplest to get the algorithm running using a data structure allowing for traditional matrix interactions (for example, individually getting and setting elements by row and column index). The available data types for working with Spark RDDs and LAPACK (Vectors and Arrays) do not support multi-dimensional formats and so I opted to build my own. I am sure, however, that my matrix class – which requires Scala for-loops – cannot be as efficient as the built in types. Due to time constraints, I was unable to refactor the algorithm with better data structures.

Rank	Lambda	Iterations	pALS Training Time (ms)	sALS Training Time (ms)	pALS RMSE	sALS RMSE
8	1	2	18434	4307	3.76	1.34
8	1	20	158058	6185	3.76	1.36
8	10	2	15933	3154	3.76	3.76
8	10	20	154755	5711	3.76	3.76
12	1	2	22103	3446	3.76	1.34
12	1	20	207937	7063	3.76	1.36
12	10	2	23496	3152	3.76	3.76
12	10	20	206648	6692	3.76	3.76

Table 1 - Performance of ALS routines on MovieLens dataset.

Conclusion

I believe *pALS* is a good start for a first attempt at writing in Scala and using Spark. Clearly, the main factor limiting my ability to complete an optimal algorithm in the allotted time is my familiarity with the paradigm for conducting linear algebra in the language. I would be very excited to learn more about best practices for handling matrix data in this framework.

Were I to continue on my own, the next logical step would be to establish timing benchmarks for the individual segments of the code. Once the most time-consuming steps have been identified, it will be straightforward to deduce the computation bottlenecks and design code that will eliminate those bottlenecks.