Large Scale Recommender Systems Using Spark

Kirk Hunter Mrunmayee H. Bhagwat Swetha Reddy

College of Arts & Sciences, University of San Francisco

March 7, 2016

I. Abstract

In this paper we describe our objective, methods, results and challenges in building large-scale recommender systems. The main motivation behind this project was to build machine learning algorithms in the distributed context. We used Amazon′s Simple Storage Service (S3), Elastic MapReduce (EMR) with Apache Spark to build a recommendation system for around 11GB of Yahoo! music data with over 700 million ratings[1]. In order to calculate similarity between songs using MapReduce technique, we referred to paper[2]. We also built a Locality Sensitive Hashing (LSH) algorithm that utilizes the random projection method. Additionally, we implemented Alternating Least Squares (ALS) on the full dataset to make predictions. Using the entire 700 million ratings in the training set, we achieve a root mean square error of approximately 1.12.

II. Introduction

The complexity of a simple collaborative filtering algorithm grows quadratically with the number of users and items in the data set. Then how can companies like Netflix and Facebook successfully implement recommendation systems when the user base grows into the millions and/or billions, much less build an effective one? The solution is to implement a recommendation system using the MapReduce framework which allows one to process data in parallel. In this paper we discuss several techniques for building large-scale recommender systems as well as their effectiveness as measured by calculating the root mean squared error under various real-world circumstances.

III. Overview and Motivation

The objective of our project was to implement a neighborhood-based and item-based collaborative filtering algorithm at scale to ultimately generate top-N music recommendations for a given user. Item-based collaborative filtering recommendation systems generally involve an intermediate step to calculate the similarity between each pair of items. The computational complexity for this approach to measure similarity is unfortunately $O(n^2)$ which increases quadratically with items/users. As a result, this process quickly becomes infeasible to implement on a single machine. Thus, we parallelized the operations using Spark, which utilizes a data abstraction that they call a resilient distributed dataset (RDD) that operates on partitioned data and performs expensive computations mostly in memory. Fortunately, there are methods for reducing the computational complexity of these operations such as alternating least squares and locality sensitive hashing, which we discuss later in this paper.

IV. Data

The dataset we have used to implement and evaluate these various techniques represents Yahoo! Music[1] community's song ratings dataset. It contains over 717 million ratings of 136,000 songs given by 1.8 million users of Yahoo! Music services. The data was collected between 2002 and 2006. Each song in the dataset is accompanied by artist, album, and genre attributes. The mapping from genre id's to genre, as well as the genre hierarchy, is given.

The data that we primarily used in our implementations is the song ratings given by various users. The files contained records in the following represented in Table I.

Yahoo! has divided the data into 10 training and 10 testing files. Each training file is approximately 1GB in size, while a given test file generally is around 30MB.

As processing large train files increased execution time, we further divided the training files to include a few million ratings for simple test when needed.

TABLE I
SAMPLE RATINGS FILE

User Id	Song Id	Rating
0	166	4
0	2245	4
0	3637	4
0	5580	4
0	5859	4

V. EXPLORATORY DATA ANALYSIS

The training dataset contains 1.8 million unique users, approximately 136,000 unique songs, and the testing dataset also contains 1.8 million unique users as well as 136,000 unique songs (Yahoo! gave data with 100% overlap between training and testing users and items for ease of analysis). The user with the most ratings gave a total of 131,000 of them. Under traditional collaborative filtering, this user would give rise to $\binom{131,000}{2}$ calculations, slowing down computation time significantly.

TABLE II
SONGS WITH HIGHEST NUMBER OF RATINGS

Song Id	Number Of Ratings
72309	35,682
105433	33,954
22763	33,794
123176	32,393
36561	32,074

TABLE III
RATINGS PER GENRE

Genre	Number Of Ratings
Unknown	65,499,138
Rock	5,293,264
Pop	1,744,753
R&B	1,295,667
Country	567,461

VI. TOOLS

A significant part of the motivation behind this project was to learn and explore different tools to process large datasets. Performance of local machines degrades as the size of the data increases. Since our goal was to build a recommendation system that scales well with an increasing number of users and items, we required a tool with very high performance capabilities. Amazon Web Services (AWS) is the most commonly used tool by industries for this purpose. This was a major reason why we decided to implement our recommender systems using AWS. The specific tools used within AWS are documented below.

Storage:
Amazon's Simple Storage Service (S3) is a highly secure, durable and scalable cloud storage service that provides its users with a simple web service interface to store and retrieve any amount of data. There is no setup cost and user pays only for the storage actually used. It was very easy to store and access train and test files from S3.

More information can be found on S3.

Computation:
Apache Spark is an open source cluster computing framework for large scale data processing. In particular, we used PySpark which is a Spark programming model for Python. Within Spark we utilized the machine learning library MLlib.

Amazon's Elastic MapReduce (EMR) web service simplifies big data processing. It is a collection of Elastic Compute Cloud (EC2) instances with Hadoop installed and configured on them. EMR provides an easy integration to run Apache Spark and smoothly interacts with data stored on S3. It served our purpose of using MapReduce to handle a vast data. Along with very high computational power, EMR is very secure and reliable. Moreover, we were able to create spot instances which were available at a very affordable rate.

More information can be found on EMR and EC2.

VII. ITEM-ITEM SIMILARITY

Item-based collaborative filtering is one of the popular recommendation systems. It calculates similarity between different items using the explicit or implicit ratings and recommends an item to the user that is similar to the items the user has been associated with.

In the context of this paper, item means a song. Thus, the recommendation system recommends a given user a song that is similar to the songs the user has listened to previously.

We used Jaccard similarity, Cosine similarity and Pearson correlation coefficient as the metrics to calculate similarity between the songs. Then we computed the item-item similarity using two approaches. Each approach is implemented using the parallel computation supported by Spark.

MapReduce Algorithm implementation from paper[2]:
Initially, we parallelized the item-item similarity computation using Spark, based on the ideas discussed in the paper[2]. But the complexity of this algorithm proved to be $O(n^2)$, resulting in the quadratic runtime as ratings increase. Hence we proceeded with using Locality Sensitive Hashing for reducing the complexity to $O(n)$.

Locality Sensitive Hashing:
Locality Sensitive Hashing (LSH) is a dimensionality reduction technique to compute similarity between a pair of items. It hashes sparse list of users into short signatures, while preserving similarity between the

items. Hash functions are used to group items into buckets such that the items in the same bucket have a high probability of being similar. In this project we implemented LSH using two approaches which use different metrics to calculate similarity. Jaccard similarity using the min hashing technique and Cosine similarity based on random projection method.

Cosine Similarity: Random Projection method of LSH calculates Cosine similarity between pairs of items. The paper[3] and the book[4] were referred to implement this algorithm. Sparse vectors of user-items ratings were used. The method is described in Algorithm 1. The time required to calculate similarity for increasing number of users using 50 and 100 vectors is captured in Fig 1.

Algorithm 1 Locality Sensitive Hashing (LSH) - Random Projection

Output: C
Input: $U_{n,1}, I_{p,1}$
Initialize k random vectors V of n-dimension and unit length;
for $i = 1,...,n_p$ **do**
  for $j = 1,...,n_k$ **do**
    $H_i \leftarrow I_i \cdot V_j$
  end for
end for
for $i = 1,...,n_p$ **do**
  for $j = 1,...,n_p$ **do**
    if $i < j$ **then**
      $C_{i,j} \leftarrow cos(Hamming\ distance_{i,j} * \pi)$
    end if
  end for
end for

Jaccard Similarity: To implement this method, we referred to the method for described in the book[4]. The list of users was converted into a sparse vector representation format. The ratings were converted to 1 and 0, based on if the user had rated the item or not. The process is described in Algorithm 2.

The main idea being if two items have the exact same minhashes in a band, they will be hashed to the same bucket and will be considered candidate pairs. Tuning the parameters namely the number of bands(higher the better), rows per band (lesser the better) and number of hash functions (higher the better), improves accuracy. We referred to the code on github[5] for this implementation. The Fig. 2 captures the runtimes for this process. One can conclude that the computation time is linear with increasing ratings. For the project we chose the number of bins into which data is hashed = 1000, number of times to hash = 1000 and number of bands = 25. It was observed that as the number of ratings increased the number of buckets/clusters increased linearly.

Algorithm 2 Locality Sensitive Hashing using minhash

Input:
Sparse vector of Users
p: Max size of the Sparse vector
m: Number of bins to hash data into
n: Number of times to hash the individual elements
b: Number of bands in which to split the signatures.
c: Minimum allowable bucket size
Steps:
1: Generate minhash signatures of input sparse vector.
2: Split signatures into bands with equal number of rows.
3: Similar minhashes in a band will be clustered.
4: Compute Jaccard similarity score of each bucket.
Output: Similar item buckets with scores.

VIII. Prediction

Item-based collaborative filtering systems predict ratings for a user-item pair using item-item similarity. We used Alternating Least Squares (ALS) algorithm from Spark's MLlib library and all 700 million training ratings to predict user ratings in the testing set. It is described in Algorithm 3.

Algorithm 3 Alternating Least Squares (ALS)

Output: U, M
Initialize M with random values;
while *Stopping criteria have not been satisfied* **do**
  for $i = 1,...,n_u$ **do**
    $u_i \leftarrow A_i^{-1} v_i$;
  end for
  for $j = 1,...,n_m$ **do**
    $m_j \leftarrow A_i^{-1} v_j$;
  end for
end while

A major benefit of using ALS on large datasets is that it can easily be parallelized, as each of the columns in U and M can be computed in parallel.

IX. Results

In Fig 1 we verify that the run time under LSH increases linearly with the number of ratings in the training set. It shows the run time to calculate item-item similarity using 50 and 100 vectors for increasing number of ratings. An increase in the number of vectors by 1 means 1 more calculation for all items. Thus, time required for 50 vectors is exactly half that required for 100 vectors. To calculate these run times, 6 cores were used on AWS with 32 partitions of the data.

Below are additional figures documenting our evaluation of the various aspects of each method used.

The Fig 2 captures the run time vs number of ratings for computing similarity using Jaccard distance. The parameters used to run the LSH were p=65,538, m=1000, n=1000, b=25, c=2. From the plot it is evident that

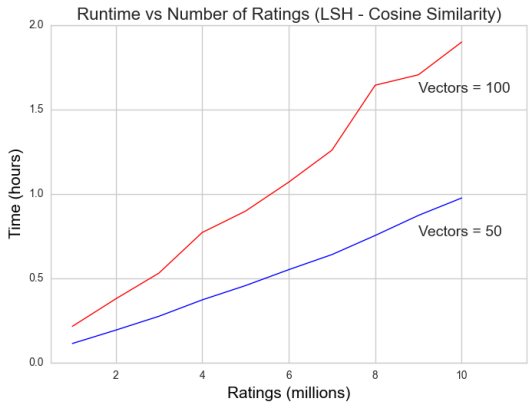

Fig. 1. Runtime to calculate item-item Cosine similarity rusing LSH.

there is a linear relationship between time taken with increasing user base.

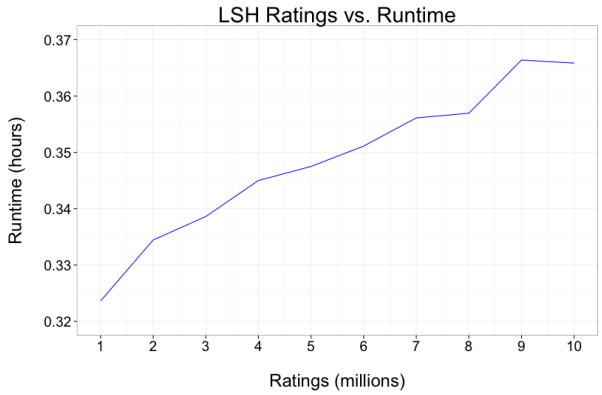

Fig. 2. Runtime to calculate item-item Jaccard similarity using LSH.

Figure 3 demonstrates the decrease in RMSE as the number of ratings increases from 100 million to 700 million. It appears as though the percentage decrease in RMSE has a logarithmic relationship with number of ratings in the training set.

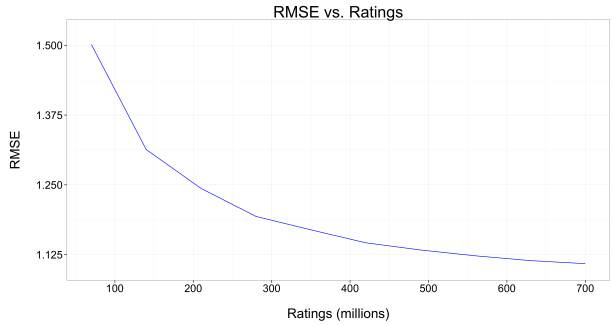

Fig. 3. Root Mean Square Error with increasing number of ratings.

In Fig 4 we explored the affect on RMSE as the percentage overlap between training and testing users takes on values of 50, 70 and 90 percent. The results coincide with our a priori expectations. Since a greater overlap between training and testing users generally means more information about item-item similarities is provided, it is not surprising that with 90 percent overlap between training and testing users we observe that RMSE is minimized.

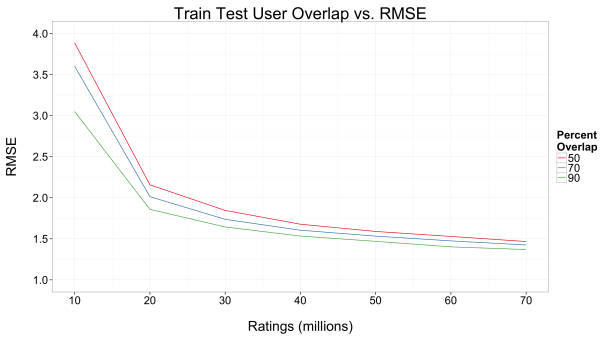

Fig. 4. Root Mean Square Error with varying degrees of train/test user-overlap.

And in figure 5 the number of ALS is increased from 1 to 25 and the decrease in RMSE is documented. It is evident from the plot that with anything beyond 5 iterations, the decrease in RMSE is negligible.

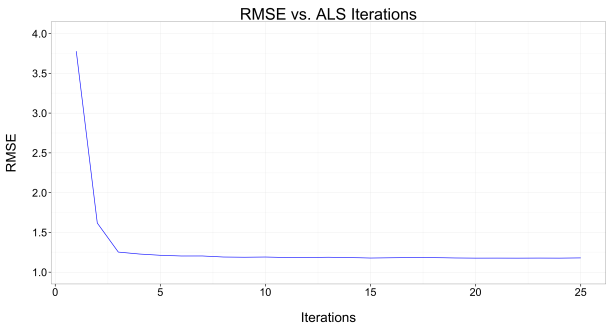

Fig. 5. Decrease in RMSE with increasing number of iterations.

X. Challenges

Handling massive data set using parallel computation algorithms was quite a challenge. Reducing the time complexity of the algorithm from quadratic to linear was also not so straightforward, even with the explicit algorithms for each method. Despite many research papers describing LSH, we were not able to find any out of the box implementations in Spark Machine learning library. This area is still being developed within Spark.

XI. Conclusion

It is important not only to be able to implement the machine learning algorithm that is appropriate in a given context but also to have the ability to use parallel computing techniques to effectively handle large amounts of data that would otherwise not be able to

handled effectively by a single machine.

For relatively large datasets, ALS should be a sufficient algorithm in order to implement a recommender system. Although LSH also allows for a linear increase in runtime with an increasing number of users, ALS is more tunable, and is preferable in many cases for these reasons. However the idea of LSH could be used in contexts of finding similar items, since it scales well.

For extremely large datasets, the open source software Apache Giraph has been developed by Facebook for their use-cases. This software utilizes a graph framework to distribute computations and it has proved to give consistently faster runtimes than ALS.

REFERENCES

[1] R2 - Yahoo! Music User Ratings of Songs with Artist, Album, and Genre Meta Information, v. 1.0 (1.4 Gbyte & 1.1 Gbyte) https://webscope.sandbox.yahoo.com/catalog.php?datatype=r.

[2] Sebastian Schelter, Christoph Boden, Volker Markl. Scalable Similarity-Based Neighborhood Methods with MapReduce http://dl.acm.org/citation.cfm?id=2365984.

[3] Deepak Ravichandran, Patrick Pantel, Eduard Hovy Randomized Algorithms and NLP: Using Locality Sensitive Hash Functions for High Speed Noun Clustering https://www.cs.cmu.edu/~hovy/papers/05ACL-clustering.pdf.

[4] Jure-Leskovec, Anand Rajaraman, Jeffrey Ullman. *Mining of Massive Datasets*.

[5] PySpark implementation of LSH using Jaccard similarity. https://github.com/magsol/pyspark-lsh.

[6] M. Winlaw, M.B. Hynes, A. Caterini, H. De Sterck. Algorithmic Acceleration of Parallel ALS for Collaborative Filtering: Speeding up Distributed Big Data Recommendation in Spark. *Department of Applied Mathematics*, University of Waterloo, Canada. August 2015. https://arxiv.org/abs/1508.03110v3