Applicability of MAHOUT for Large Data Sets

Experiences and Lessons Learned

Michael Sevilla
University of California, Santa Cruz
1156 High Street
Santa Cruz, California 95064-1077
msevilla@soe.ucsc.edu

ABSTRACT

In an attempt to unify two hot trends in the research community, this project explores distributed data mining. The idea of taking big data and processing it to extract information on a large number of nodes has many applications for many sectors outside of computer science. This project explores Apache Mahout and attempts to quantify its modifiability, accuracy, and performance. To determine the modifiability, the code is opened up and examined. To get an idea of the accuracy, the resulting output is scored and compared against the highest-scoring algorithm in an online competition for the same dataset. Finally, the performance is examined to see how well Mahout scales.

This project shows the importance of understanding the fundamentals of any data mining solution before attempting to use it. Poor parameters and a general lack of understanding led to interesting results and, more importantly, a number of useful "lessons learned".

1. INTRODUCTION

Data mining, as a field, has many applications for real-world systems and data. The construction and implementation of algorithms for processing large amounts of data is being conducted for both business and hobby. The idea, from the Kaggle website

The idea for this project is to show how much better distributed data mining is when compared to data mining on a single node and to quantify the sheer power of distributed computing. The initial hypothesis is that more machines, no matter the algorithm, is always better than one. This project proves that this hypothesis is flawed for a variety of reasons.

The hypothesis is that 22 machines and sophisticated, crowdsourced data mining libraries will outperform one machine running a small collection of python code. Mahout’s modifiability, accuracy, and performance are examined by looking at one of its distributed data mining algorithms - The Collaborative Filter Recommendation Job.

2. METHODOLOGY

To test the hypothesis, a series of tools, statistical models, and applications needed to be explored. Section 2.1 discusses the existing tools that assisted the exploration of distributed data mining. Section 2.2 explores the collaborative filtering algorithm.

2.1 Background

This section discusses the existing applications, data sets, and resources that are used in this project.
2.1.1 Mahout

Mahout [6] is a set of distributed data mining libraries that interface with an underlying distributed system. The framework for the distributed system is Hadoop, which implements MapReduce. Mahout is a said to be scalable for large data sets, although there were not quantifiable measurements or definable statements in regard to what this actually means [6]. Furthermore, the Mahout community is highly collaborative and reasonably well documented. The code is contributed by programmers from all different computer science, data mining, and statistics disciplines.

Mahout already has support for two collaborative filtering algorithms - one for item-based similarity and another one for job recommenders. This project analyzes the the Mahout job recommender algorithm because of the available comparison points in the community.

2.1.2 www.kaggle.com

Kaggle [5] is a website that aims to bring the hobbyists to the front of the data mining community by providing public competitions to solve real-world problems. The goal for one of its competitions, the Kaggle Million Song Dataset Challenge, is to recommend songs to a user using a large data set and the user’s previous listening history. Technically, the goal is to recommend 500 songs for all 11,000 users. The efficiency and performance of the algorithm is measured with a statistical scoring system. The Kaggle Million Song Dataset homepage has additional resources, a forum, and a FAQ section that aided in the design and implementation of this project.

Martin L [7] is a Kaggle user who provided a solution and working code to promote collaboration and inner-community data mining research. He was the number one leader on the leaderboard in the early stages of the competition after submitting a simple algorithm and an even simpler implementation. Much of the code is incompatible with the data set that I used and had to be modified to accommodate this format. Martin L’s algorithm is used for comparison purposes throughout this report.

2.1.3 The Data Set

The purpose of the data set is to encourage research in the field, provide a benchmark data set for experiments, provide a large, accessible data set for the community, and to improve researchers collaboration. This data has been ported to Amazon Web Services and is accessible to all Mahout’s users in a readily available medium.

The full data set is humongous - it has metadata for over 1 million songs that tallies up to 280GB of data. The data set has over a million users, 300,000,000 unique songs, and 48 million <user, song, count> triplets. This data set is open source and can be pulled from the echo-nest repository. For this project, only a subset of the data is operated on.

2.2 Collaborative Filtering

The goal of collaborative filtering is to predict an item for a user. The algorithm consists of three steps to construct a top-n-recommendation system.

1. Parse the input data
2. Construct user-item-matrix
3. Predict missing entries ...

For the data set, parsing the input data consists of reading in the input <user, song, count> triplets, performing some pre-processing to reduce the size of the input, and outputting a file that can easily be digested by Martin L’s solution and Mahout’s solution. This compression is a result of changing long string IDs to ints - the number of characters is drastically reduced. ints are easier to process and manipulate.

Second, a user-item-matrix [6] is constructed. The algorithms for the data matrix construction can be viewed in sections 3.1.1 and 3.1.2.

Finally, predictions are generated for the missing and future entries. This is accomplished by taking the resulting user-item-matrix from step 2 and filling in values through simple linear algebra combinations. For this project, the output is 500 recommendations for all 11,000 users.

2.2.1 Scoring the algorithms

Kaggle measures accuracy using the mean average precision on a blind data set (not released). The contest final assessment will also use average rank, precision at K, AUC, etc. [5]. For this project, since there is no validation set for the data, values are manually extracted and the recommendations are later scored using these removed values. Since these are valid values, the resulting score should also be valid. 8 or 9 values are taken out of the training set from user 2, user 16, and user 17. This smaller training data is then parsed, processed, and used to predict the 26 values that were extracted. A big flaw in this method is sample size and scope. Only 3 users are modified and compared to the resulting recommendations - which accounts for far less than the number of users that recommendations were generated for. The mean average precision (MAP) is calculated using:

$$\text{MAP} = \frac{\sum_{q=1}^{Q} \text{AveP}(q)}{Q}$$

where Q is the number of queries.

3. EVALUATION

To evaluate Mahout, tests are conducted to get a quantifiable metric for the modifiability, accuracy, and performance of the recommender algorithm. Each of these tests are performed on a Linux server 2.6.35-22. Each machine has 8GB of RAM and 4 2GHz processors.

3.1 Modifiability - The code

To examine modifiability, the code complexity of the Mahout Job Recommender is compared to Martin L’s solution.

3.1.1 Martin L’s Solution

Martin L’s solution is straight forward and easy to understand. The first step, as consistent with the collaborative
filtering for item-based recommendations, is to parse the data.

The second step is to construct the user-item-matrix. This is constructed as a sparse matrix that represents the similarity vector of the user’s history. The matrix, which is called the "colisten matrix", represents listeners who listened to songs i and j. The matrix is an n x n matrix where n is the number of songs. This "colisten matrix" represents the cooccurrences of all similarity vectors to the listening history for all users.

\[
\text{COLISTEN}_{i,j} = \begin{bmatrix}
a_{0,0} & a_{0,1} & \cdots & a_{0,n} \\
a_{1,0} & a_{1,1} & \cdots & a_{1,n} \\
\vdots & \vdots & \ddots & \vdots \\
a_{m,0} & a_{m,1} & \cdots & a_{m,n}
\end{bmatrix}
\]

Finally, once the user-item-matrix is constructed, the recommendations for each user can be computed. This is achieved by summing the row vector for every song that a user recommended or listened to. If less than 500 songs are recommended, the rest of the entries are filled in with high frequency count values, using entries along the diagonal.

The python code is very concise and well constructed. The code sums to 118 lines of code. The minimal code was not optimized by what it lacked in speed it made up for in clarity. The concise nature of the syntax and the usage of python’s most intimate libraries drastically reduced its size and complexity. As a result, I could actually hook in my own code for parsing triplets and return values that his code expected.

3.1.2 Mahout’s Solution
Mahout’s code is far more complex and did not lend easily to modification for possible performance increases. The density of the code and the necessity to understand the MapReduce framework prevented me from making changes to Mahout to leverage latencies. The only insight for the code behavior is from the Mahout Webpage [6], which provides a nice high level overview but little detail.

There are three main components in the Mahout source code that need to be edited in order to modify the underlying algorithms. The files to be edited are the RecommenderJob.java, MatrixRowWrapper.java, and the UserVectorizerSplitterMapper.java and the associated lines of code for these code files is shown in figure 4.

For the purposes of this project, Mahout is treated as a black-box.

From a programmer’s view, Martin L’s solution is far easier to understand, utilize, and modify than Mahout’s Recommender Job.
3.2 Accuracy - Small Data Set
To examine accuracy, the Mahout Job Recommender is compared against Martin L’s solution using the scoring metrics provided by Kaggle.

A small subset of the data is processed to find 500 recommendations for about 11000 users. The data, kaggle_visible_users.txt, is 86MB and consists only of <user, song, count> triplets. Kaggle also provides a kaggle_user.txt and kaggle_songs.txt for verification purposes. This is supposed to be the validation data but was used for recommendation filtering in this small experiment.

3.2.1 The Accuracy of Martin L’s Solution
Martin L’s code performed well on one machine and outputted the top score for the Kaggle competition. The code on the Linux server took an hour and 50 minutes to produce the 500 recommendations for each user. Parsing the data and creating the colistening matrix is fast - only about 32 minutes. Most of the time is spent, not in the user-item-matrix construction, but in the song predictions for 11000 users. This is very different than Mahout’s performance and scalability improvements, which is addressed in the next section.

The accuracy and the system is validated by the Kaggle leaderboard - the algorithm achieved the top score and was measured at 0.0174 for 9% of the data [5]. Figure 5 shows the output from the above test. From this, the following scores are calculated:

\[
\text{User2} = \frac{\sum_{q=1}^{Q} \text{AveP}(q)}{Q} = \left( \frac{1}{3} + \frac{2}{7} + \frac{3}{12} + \frac{4}{28} + \frac{5}{40} \right) = 0.074
\]

\[
\text{User16} = \left( \frac{1}{9} + \frac{5}{120} + \frac{3}{314} \right) = 0.005
\]

\[
\text{User17} = \left( \frac{1}{8} \right) = 0.004
\]

3.2.2 The Accuracy of Mahout’s Solution
The same subset of song triplets is also processed using Mahout’s Collaborative Filtering recommender job. The algorithm ran in about 30 minutes, with a majority of the time spent creating the matrix jobs. The small data set was not meant to prove performance gain, since 22 machines should easily be able to analyze 86MB data.

The accuracy is similar to Martin L’s performance:

\[
\text{User2} = \frac{\sum_{q=1}^{Q} \text{AveP}(q)}{Q} = \left( \frac{1}{3} + \frac{2}{7} \right) = 0.013
\]

\[
\text{User16} = \left( \frac{1}{9} + \frac{5}{120} + \frac{3}{314} \right) = 0.0010
\]

\[
\text{User17} = \left( \frac{1}{8} \right) = 0.004
\]

The scores are reasonably close to Martin L’s scores. A more reasonable test would take into account a much higher percentage of the data.

Figure 5: The output used to determine the scores of a list of songs for a user using Martin L’s code.

Figure 6: The output used to determine the scores of a list of songs for a user using Mahout’s code.

Despite the fact that the cluster implementation with Mahout had more resources and computer power, the algorithms still performed equally well.

3.3 Performance - Larger Data Sets
Performance for Mahout’s Collaborative Item-based Filtering is surprising, especially given the intended use of the system. Performance measures are only taken on Mahout for differing data sets, ranging from 64MB to 3GB. It was the hypothesis that the Mahout Recommender job would have no problem processing this data, especially since Hadoop should scale well to many machines. The mahout job is run using the defaults:

mahout recommenditembased –input input3G/mInput.csv
–output recKaggle-3G-thresh2 –usersFile input3G/users.txt
–similarityClassname SIMILARITY_COCURRENCE

To examine the performance, the Mahout Job Recommender is tracked for varying input data sizes. To measure the performance, time is used on the order of minutes and seconds. This offers a discernable granularity between tests, as tests ranged from seconds to hours. Units for total test time are scaled to minutes and all other tests are measured in seconds. The time measurements for are taken from Hadoop’s browser interface using a python script.

Figure 7 shows the scalability of the Job Recommender measured by job completion time. This is a macro-level view and shows total completion time in seconds. The initial trend of the curve looked promising, as larger and larger files are processed. At 512MB, it looked like the curve would level out and that there would be obvious performance benefits for Mahout. Contrary to this prediction, the time started to scale linearly when the data set sizes started getting larger.
Figure 7: Initial results indicate that Mahout does not scale well for normal sized jobs. At 3GB, the MapReduce started failing nodes and the job had to be killed. Upon further review, it is determined that these poor results are a consequence of poor job specifications and a small bug in the Mahout code base.

then 512MB. At 3GB, Mahout started failing nodes and at the time that the trace is stopped, 10 nodes had failed. The total times also started to become infeasible. For example, a two hour job for 1GB data on a cluster of 22 machines is not a predictable result.

This results warranted further inquiry, so measurements are also pulled for each Mahout jobs. Each job is automatically segmented by Mahout into 3 categories: preparation, row similarity calculations, and recommendation predictions.

Figure 9 shows the preparation phase of the recommender job. Each task is conducting pre-processing to map ID indices to vectors and preferences. These all seem to scale well with time, all completing on the order of minutes. This does not appear to be the bottleneck.

Figure 10 shows the row similarity computations for each job. Two of the jobs scale very well, namely the construction of the cooccurrence matrix and the vector normalization. The unsymmetrify function takes a large amount of time.

The unsymmetrify function performs poorly because of the size of the record and a large number of cooccurrences in the data. Some items are occurring too frequently with a large number of other records, resulting in many non-similar items in 1 record. To address this problem, the Mahout Recommender has a threshold value that can be set at initialization. The threshold filters out values that fall below a certain similarity value - these items are not processed by the recommender.

To solve the problem, the samples need to be filtered by reducing the threshold. In an attempt to discard unsimilar data, the threshold values were changed from:

```
-- threshold 0.1
-- threshold 0.5
-- threshold 10
```

Changing the threshold to 10 caused a noticeable speedup for the unsymmetrify task. The job took only took 19 minutes and 30 seconds, as indicated by the red X in figure 11. This is a true testament to the need for a good understanding of the algorithm that you are using.

Browsing the Mahout bug reports shows that the problem with unsymmetrify is a combination of human error and a small bug in Mahout. The Mahout implementation error does not modify the threshold correctly, which explains why I was not seeing any results. A patch has been developed to fix this issue [4].

Finally, figure 8 shows the scalability of the recommender jobs. These perform remarkably well for larger data sets, especially when comparing them to Martin L’s solution. This is encouraging for recommender systems because once the initial user-item-matrix is constructed, further additions and predictions can be very fast. It appears that the performance slowdown between 128MB and 256MB is an outlier - possibly due to the small data size.

These tests did not prove that Mahout is not scalable - rather that Mahout is not as intuitive as previously predicted. A Mahout user must have intimate knowledge of the underlying algorithm in order to parameterize the algorithms cor-
4. LESSONS LEARNED

This project taught me a great deal about the importance of methodology. In order to be a good data miner, you have to (a) understand the data, (b) understand the underlying algorithms, and (c) avoid adding unnecessary resources to the system.

A poor understanding of the data prevented me from fully leveraging the algorithms and their positive tradeoffs. For example, the Million Song Data Set is highly structured and slightly repetitive. This leads to problems with the cooccurrence matrix and low similarity values. Intimately understanding the data would have helped narrow down the problem to that of a user error instead of quickly jumping to the conclusion that Mahout has a poor implementation.

A proper understanding of the underlying algorithm would have lead me to immediately analyze the proper tasks and I could have determined where and why I was getting such poor performance. When I witnessed this poor performance I did not know how to proceed because I did not completely understand what matrix the algorithm was calculating and what poor parameter choices might entail. In order to properly use and implement an algorithm, I need to know the theory behind it.

Finally, I should not have attempted to continue to add resources to a system that did not need them. Coming from a systems background, I am used to continually scaling out a system until it solves the problem. I tried this methodology again with this project and wasted valuable time and effort. Understanding what resources I had available and what I could do with them proved to be a much better strategy and ultimately helped me get back on the right track.

The project can be deemed a success because of the process and the lessons learned. Although the results are beneficial and interesting, the process that I took to reach the conclusions and analysis is far more beneficial.

5. CONCLUSION

This project attempts to demonstrate the power of a distributed systems, but instead proves that a lack of data mining awareness can lead to unpredictable results. In an attempt to quantify Mahout’s modifiability, accuracy, and performance, the Mahout distributed item-based collaborative filter recommender job is run over the Million Song Data set. Contrary to the hypothesis, Mahout is not as easy-to-use “out of box”. Large directories and source files detract from Mahout’s modifiability. The accuracy is comparable to a strong algorithm operating on a single machine. The performance does not scale well if the system is not configured correctly for the given data set and algorithm. Despite the fairly negative results and user-experiences, the resulting project still has interesting conclusions and enough analysis to encourage the possibilities of an extension for further experimentation on the system.

6. REFERENCES