Dimension Independent Cosine Similarity for Collaborative Filtering using MapReduce

By
Mr. Fei Shen

Submitted in Partial Fulfillment of the Requirement for the Degree of Master of Science in Computer Science Assumption University
December 2015
Dimension Independent Cosine Similarity for Collaborative Filtering using MapReduce

By

Mr. Fei Shen

Submitted in Partial Fulfillment of the Requirement for the Degree of
Master of Science
in Computer Science
Assumption University

December 2015
Vincent Mary School of Science and Technology

Master Thesis Approval

Thesis Title: Dimension Independent Cosine Similarity for Collaborative Filtering using MapReduce

By: Mr. Fei Shen
Thesis Advisor: Dr. Rachsuda Jiamthaphaksin
Academic Year: 1/2015

The Department of Computer Science, Vincent Mary School of Science and Technology of Assumption University has approved this final report of the twelve credits course, SC7000 Master Thesis, submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Approval Committee:

(Dr. Rachsuda Jiamthaphaksin)
Advisor

(Dr. Anilkumar Kothalil Gopalakrishnan)
Committee Member

(Dr. Songsak Channarukul)
Committee Member

(Asst.Prof.Dr. Pirawat Watanamongse)
Commission of Higher Education
Ministry of Education

Faculty Approval:

(S. Chun)
(Dr. Songsak Channarukul)
Program Director

(Asst. Prof. Dr. Jirapun Daengdej)
Dean

December 2015
ACKNOWLEDGEMENT

I would like to express my most sincere appreciation and thanks to my advisor, Dr. Rachsuda Jiamthapthaksin for her encouragement and advice through the course of this study. Her constant guidance in algorithm and experiment design helped me in completing this study. Her patiently review and comments helped in writing of this thesis.

Special thanks to Dr. Songsak Channarukul, Program Director of Master of Science in Computer Science and committee member, Dr. Anilkumar Kothali Kopalakrishnan and Asst. Prof. Dr. Pirawat Watanapongse for reviewing my research work.

I also would like to thanks to all the faculty members of Department of Science and Technology for their help and encouragement throughout my study.
ABSTRACT

DIMSUM, an efficient and accurate all-pair similarity algorithm for real-world large scale dataset, tackles shuffle size problem of several similarity measures using MapReduce. The algorithm uses a sampling technique to reduce ‘power items’ and preserves similarities. This work presents an improved algorithm DIMSUM+ with a complex sampling technique to enhance DIMSUM so that it is able to further reduce ‘power users’. The algorithm generates $k$-nearest-neighbor matrix that is used in collaborative based Recommender systems. The evaluations of algorithm on MovieLens dataset with 1 million movie ratings and Yahoo! Music dataset with 700 million song ratings show significant improvement that DIMSUM+ outperforms DIMSUM at least 1.4x faster.
# Table of Content

ACKNOWLEDGEMENT \hfill i

ABSTRACT \hfill ii

LIST OF FIGURES \hfill v

LIST OF TABLES \hfill vi

CHAPTER 1 \ INTRODUCTION \hfill 1

1.1 Background \hfill 1

1.2 Motivation \hfill 2

1.3 Item-based Collaborative filtering \hfill 3

1.4 Problem Definition \hfill 4

1.5 Goals and Objectives \hfill 5

CHAPTER 2 \ THEORITICAL BACKGROUND \hfill 7

2.1 Hadoop MapReduce \hfill 7

2.2 Cosine Similarity using MapReduce \hfill 8

2.3 DIMSUM Cosine Similarity \hfill 9

2.4 Mahout Cosine Similarity \hfill 11

CHAPTER 3 \ LITERATURE REVIEW \hfill 12
LIST OF FIGURES

Figure 2-1  Dataflow in MapReduce from "Hadoop: The Definitive Guide" by Tom White [12]  

Figure 5-1  MAE surface with various \((p, s)\)  

Figure 5-2  Prediction quality (MAE) and computational time for different \(p\) values in the small dataset  

Figure 5-3  Prediction quality (MAE) and computational time for different \(p\) values in the large dataset  

Figure 5-4  Comparing DIMSUM and DIMSUM+(\(p=500\)) in the small dataset  

Figure 5-5  Prediction quality and runtime for different \((p, s)\) values  

Figure 5-6  Prediction quality and shuffle size for different \((p, s)\) values  

Figure 5-7  Linear scale with number of users with trend lines showing different slopes  

Figure 5-8  Recall for algorithms using different top \(N\) recommendation
LIST OF TABLES

Table 5-1  Test parameters

29
CHAPTER 1
INTRODUCTION

1.1 Background

With Internet technology nowadays companies, organizations and people in the world have been generating extremely large amount of information in every millisecond. For example, in each minute Google receives approximately 4 million search queries while Facebook users share approximately 2.46 million contents [1]. Recommender systems are one of the applications that provide Internet users an ability to find needles in the haystack. It suggests products/services based on user preferences. The rapidly growing of data size brings challenges to development of recommender systems as well.

*Collaborative Filtering* is one of the most popular approaches used in Recommender systems. It relies solely on interactions between users and items in the past. *Model-based collaborative filtering* algorithms, which use various matrix factorization methods, often give high prediction quality. Alternative least square is one of the finest parallel matrix factorizing methods on distributed systems to handle gigantic data size [2, 3]. However, it is an inevitable problem that the iterative implementation, specifically in Hadoop [4], suffers from the initialization overhead of distributed
frameworks. An alternative approach is *memory-based collaborative filtering*. It firstly finds the *k*-nearest-neighbors (*k*-NN) of an item or user, and then uses the neighbors’ interaction history in order to make predictions. Since the process of finding *k*-NN for all users/items usually takes a long time, it is often preferable to use the *item-based similarity matrix* [5], which processes *k*-NN offline so that it does not require online calculation for individual incoming users.

1.2 Motivation

A common but effective technique used to deal with large size of data is sampling. However, applying down sampling always increases data sparsity, which degrades prediction accuracy. In order to improve efficiency and to prevent accuracy loss caused by sampling, researchers start targeting ‘power users’ or ‘power items’. When a movie is popular, it usually receives feedback/reviews from users more than average. Consequently, in similarity computation other movies tend to compare to all feedbacks given to this movie, thus causing tremendous computation overhead. The same situation also occurs to popular users. There are research works dealing with either ‘power users’ or ‘power items’ [6, 7, 8], but none of them deals with both targets. The motivation of this work is to design and to develop a sampling-based similarity
computation algorithm that takes into account both ‘power users’ and ‘power items’, which enhance computational performance.

1.3 **Item-based Collaborative filtering**

Collaborative filtering (CF) is a process of predicting user’s preference for an item that is new to him/her by using his/her historical preference on other items. The preference value can be an explicit feedback from a user, for instance, ‘like’ or ‘dislike’ and 5 Likert scale’s rating. It can also be an implicit feedback extracted from user’s actions, for instance, web browsing history.

Normally a limited set of users or items is chosen for the prediction of one user-item pair. The set of $k$ nearest neighbors ($k$-NN) is defined as the first $k$ users (items) that are the most similar to the target user (item). There are two basic approaches in collaborative filtering that apply $k$-NN: user-based using users’ $k$-NN and item-based using items’ $k$-NN. Both approaches are similar in such a way that two matrix need to be multiplied to generate the similarity matrix. However, item-based approach is more popular than user-based approach due to its ability to cope with new users; therefore, this work focuses on the item-based approach.

According to our observation, for *item-based collaborative filtering* [5] the
similarity calculation and neighbor selection are the most time consuming tasks. Therefore, they are usually processed offline, and then the rating prediction and recommendation can be computed either as online processing or as batch processing. The rating prediction can be calculated by rating aggregation of $k$-NN, whose detail is given in section 1.4.

1.4 Problem Definition

Let $A$ be a rating matrix having size $|U| \times |I|$, where $U$ is a set of users, and $I$ is a set of items. Let $a_{ui}$ and $a_{ui}$ denote rating vectors of user $u$ and item $i$, respectively. Accordingly, $a_{ui}$ denotes a rating given to item $i$ by user $u$. The matrix $A$ is usually a sparse and skinny matrix; in other words, $|U| \gg |I|$, and total number of ratings is much less than $|A|$. For a large dataset, it is feasible to store the matrix on single machine, but it is impossible to load the whole matrix into main memory. The item-to-item similarity matrix can be defined as $S = A^T A$. The problem is formulated as finding the $k$-NN matrix, which is a subset of $S$, where each row and column has at most $k$ largest elements. Traditional approaches that require random access to matrix $A$ cannot work efficiently when the dataset is too large to fit in main memory.

The similarity values can be computed by various measures. This work makes use
of cosine similarity, which is popularly used with CF. The formal cosine similarity $S_{ij}$ for items $i$ and $j$ is given by (1-1). To predict rating for user $u$ and item $i$, we calculate the weighted average of ratings from $i$'s $k$-NN as specified in (1-2), where $a_{ui}$ denotes ratings of user $u$ on co-rated item $j$.

$$S_{ij} = \frac{a_{ui}a_{uj}}{\|a_{ui}\|\|a_{uj}\|} \quad (1-1)$$

$$\text{pred}(u,i) = \frac{\sum_{j \in kNN(i)} S_{ij}a_{uj}}{\sum_{j \in kNN(i)} S_{ij}} \quad (1-2)$$

1.5 Goals and Objectives

Apache Mahout and Apache Spark [9, 10] are open-source big data processing frameworks that include cosine similarity algorithms, which are capable of handling large scaled datasets. The algorithm in Apache Mahout (adhere Apache Mahout algorithm) uses ‘interaction cut’ to eliminate ‘power users’ while the algorithm in Apache Spark (adhere Apache Spark Algorithm) uses probability sampling to reduce the influence of ‘power items’. Both implementations achieve significant performance improvement. The goal of this work is to propose an improved algorithm that deals with both ‘power items’ and ‘power users’, which results in more performance improvement.

In order to accomplish that, the objectives are listed as follows:
1. The proposed algorithm will output the $k$-NN matrix which can be immediately used for rating prediction.

2. For a limited time, the proposed algorithm will produce better accuracy than the original one.

3. For a target accuracy, the proposed algorithm will require less time than the original one.

4. The runtime of the proposed algorithm will scale linearly with an increment of dimensions\(^1\).

---

\(^1\) In this case, dimension is number of users.
CHAPTER 2
THEORITICAL BACKGROUND

2.1 Hadoop MapReduce

Hadoop is an open-source implementation of MapReduce [11] paradigm for large
dataset processing. It frees developers from complex tasks by handling complex tasks,
for instance, job scheduling and failure recovery for them. It allows programs to run on
distributed computers with no shared memory, and all computational code goes into
two functions:

\[\text{map}(k_1, v_1) \rightarrow \text{list}(k_2, v_2)\]

\[\text{reduce}(k_2, \text{list}(v_2)) \rightarrow \text{list}(v_3)\]

As shown in Figure 2-1, the data on Hadoop Distributed File System (HDFS) is
firstly split into blocks and sent to mappers of all cluster machines. After a mapper
finishes all its map functions, it enters shuffle phase, where all outputs are sorted by \(k_2\),
and ensures all values with same key will go to the same reducer. Optionally, a combiner
function can be assigned after mapper works as a local reducer to reduce shuffle size.
Finally, the output $v_3$ is written to HDFS as a final result.

![Diagram](image.png)

Figure 2-1: Dataflow in MapReduce from "Hadoop: The Definitive Guide" by Tom White [12]

### 2.2 Cosine Similarity using MapReduce

Algorithm 1: Cosine Similarity by Brute Force

```plaintext
function map($u_\ast$):

    foreach $(i, j) \in u_\ast \text{ with } i < j$ do

        emit $(i, j) \rightarrow a_{ui}a_{uj}$

function reduce $((i, j), (v_1, v_2, ..., v_n))$:

    emit $(i, j) \rightarrow \frac{\sum v}{\|u_i\|\|u_j\|}$
```

Referring to many research works [6, 7, 8, 13], Algorithm 1 is considered an
acceptable way to compute the all-pair similarity in MapReduce framework. The map
function multiplies item vectors in a distributed way. The reduce function then adds up
the dot products for each item vector and normalizes each of them with their L2 norms.
This algorithm neither requires shared memory nor random access to a dataset. The
dataset can be stored in distributed fashion, and accessed in a row-by-row manner.
However, as this process requires both tremendous computational time and data
broadcasting over the network, it is clear that a sampling technique must be applied.

2.3 DIMSUM Cosine Similarity

DIMSUM provides a sampling approach for various similarity measures using
MapReduce model. It focuses on reducing the shuffle size. Its effect in reducing the
network traffic has been theoretically proven and also tested with live Twitter data [8].
Since this work focuses on cosine similarity, we use the latest version that splits the
sampler into two phases\(^2\), which reduces the total number of random generation.

DIMSUM is given in Algorithm 2.

This algorithm works in a simple but clever way using probability sampling. For
each item pair, they are down sampled by their L2 norm of their rating vectors. During

\(^2\) In DIMSUM version 1, \((i,j)\) throws a coin together, but in DIMSUM version 2, \(i\) throws a coin first. If
\(i\) wins, \(j\) then throws again. As a result, one sampler becomes two.
the sampling, the more ratings an item received, the less likely it will be chosen.

Therefore, by reducing the influence of 'power items' the network is free from being flooded by popular items. The reducer function is simply summation, so the expected output is mathematically equivalent to Algorithm 1. Zadeh et al. claim that this algorithm preserves the similarity values to be larger than $s$ when the oversampling factor is set to $\gamma = \Omega(\log(n)/s)$ [7]. The final output of this algorithm is all-pair similarity matrix.

---

**Algorithm 2: DIMSUM Cosine Similarity**

```plaintext
function map($a_{ui}$):
  foreach $i \in a_{ui}$ do
    with probability $\min \left( 1, \frac{\sqrt{\gamma}}{\|a_i\|} \right)$
      foreach $j \in a_{ui}$ with $i < j$ do
        with probability $\min \left( 1, \frac{\sqrt{\gamma}}{\|a_j\|} \right)$
          emit ((i, j) → $\frac{a_{ui}a_{uj}}{\min(\sqrt{\gamma}||a_i||)\min(\sqrt{\gamma}||a_j||)}$)
  function reduce ((i, j), (v_1, v_2, ..., v_n)):
    emit (i, j) → $\sum v$
```

---
2.4 Mahout Cosine Similarity

Apache Mahout algorithm is similar to Algorithm 1 except that it performs an ‘interaction-cut’ on user vectors. The algorithm can be described through the following steps:

1. Calculate norms for all item vectors.

2. Given an ‘interaction-cut’ of size $p$, sample down all user vectors to size $p$ if the size of the vector is larger than $p$.

3. Calculate top-right half similarity matrix by using Algorithm 1.

4. Get the full similarity matrix by merging results from the step 3 with its transpose. A priority queue is used to keep only $k$ most similar items for each item, so the final output is the $k$-NN matrix.

The ‘interaction-cut’, unlike the smart sampling of DIMSUM that preserves information, is a straightforward down sampling. For example, a user with 1,000 ratings, when $p = 500$, each rating will have 50% chance to be removed from training. The direct result is huge improvement in speed as well as data sparsity. During our tests with this algorithm, we discovered the ‘missing prediction problem’ caused by increased sparsity. The detail of the problem is given in section 5.6.
CHAPTER 3
LITERATURE REVIEW

NN-Descent [14] is a fast $k$-nearest-neighbors ($k$-NN) estimation algorithm. It is based on the assumption that the nearest neighbors of one’s neighbor is highly likely to be similar to the one. It avoids computation of full-pair similarities by exploring the neighbors of a neighbor. However, this algorithm does not work well with high dimensional data. Another famous algorithm for approximate $k$-NN search is Locality Sensitive Hashing (LSH) [15]. It requires a special designed hash function for a certain similarity measure that is cosine similarity [16]. Two similar vectors tend to have a high probability in order to have the same hash value, thus it reduces the dimensionality of data. However, achieving high accuracy requires high computation.

Schelter et al. (2012) proposed a scalable method for a generic similarity calculation on Hadoop [6]. It uses interaction-cut to reduce the overhead of ‘power users’. As a result, it scales linearly with both number of users and number of machines in cluster. The authors provided open-source implementation in Mahout, and their idea of reducing ‘power users’ inspires us in developing our work.

In addition to the implementation on Hadoop, Zadeh and Goel proposed an algorithm in 2012 that reduces shuffle size and number of reduce keys when computing
all-pair similarities using MapReduce [7]. Their algorithm uses a clever sampling to reduce 'power items' letting shuffle size independent of number of users. Zadeh and Carlsson later proposed in 2013 a revised version of cosine similarity called DIMSUM [8]. Their algorithm is proven to be accurate and works efficiently on the live Twitter data.
CHAPTER 4
ALGORITHM

This section explains the proposed algorithm DIMSUM+; in general, the algorithm adds one more sampler to the other dimension. Since the algorithm requires 2 input parameters, a procedure to find the suitable parameters for DIMSUM+ is also given.

4.1 DIMSUM+

Inspired by interaction-cut of Mahout cosine similarity as discussed in section 2.4, which eliminates the power users, we propose the DIMSUM+ with an extra sampler on users, whose detail is given in Algorithm 3.

Three modifications are made to original DIMSUM algorithm. The first one is the reduce key; instead of using item pair \((i, j)\), it uses only the item id \(i\) by storing all corresponding \(j\) in a temporary vector for the next phase. This approach is also adopted from [6]. In Hadoop, all outputs from the map function will be sorted by key before sending to reducer [12]. Sorting by single valued keys is considered much efficient than on compound keys. The other reason for this modification is the desirable output; while DIMSUM outputs all-pair similarities, DIMSUM+ only needs the top \(k\) neighbors of each item for rating prediction. Storing all neighbors in one vector provides a more
efficient way to obtain $k$-NN, which leads to the second modification; the reduce function generates outputs only top $k$ elements for each item. As a result, we have the top-right half of the $k$-NN matrix. The full $k$-NN matrix can be obtained by simply transpose-merge [6].

The last modification is adding another independent sampler to DIMSUM. After sampling by DIMSUM, items $(i, j)$ will face another coin flipping with the probability of $\frac{p}{|a_{u, i}|}$, where $p$ is predefined. According to [7, 8], Algorithm 2 reduces the shuffler size of each item pair to $O(|I|L \log |I| / s)$, where $L$ is the maximum number of ratings given by one user. When a dataset has a long tail distribution over rating count per user, we have $|a_{u, i}| \ll L$ for most users, and $|a_{u, i}| < L$ for all users. We can trivially show that our algorithm has a shuffler size of $O(p|I| \log |I| / s)$. Moreover, our algorithm is compatible with the combine function; when the combiner is applied, the shuffle size will be even lower.

Comparing with the last modification, the first two modifications are more technical than algorithmic. In order to verify the effectiveness of DIMSUM+ with minimal cross-framework difference, we re-implemented DIMSUM on Hadoop with the first two changes. Details of experiments and results are given in CHAPTER 5.
Algorithm 3: DIMSUM+ Cosine Similarity

function map($a_{u*}$):

$$\beta = \frac{p}{|a_{u*}|}$$

foreach $i \in a_{u*}$ do

with probability $\min\left(1, \frac{\sqrt{n}}{|a_{u*}|}\right) \min\left(1, \sqrt{\beta}\right)$

foreach $j \in a_{u*}$ with $i < j$ do

with probability $\min\left(1, \frac{\sqrt{n}}{|a_{j*}|}\right) \min\left(1, \sqrt{\beta}\right)$

put $(j, \frac{a_{u*}\max(1,\beta)}{\min(\sqrt{n}|a_{u*}|)\min(\sqrt{n}|a_{j*}|)})$ in $vector_i$

emit $i \rightarrow vector_i$

function reduce $(i, (v_1, v_2, ..., v_n))$:

$vector_i \rightarrow vector\_sum(v_1, v_2, ..., v_n)$

emit $i \rightarrow$ top $k$ elements in $vector_i$

4.2 Finding the Suitable Parameters

The encountered issue of DIMSUM+ is now on finding the suitable parameters.

DIMSUM uses a single parameter $s$, which is easy to find a value that balances the accuracy and executional time. On the other hand, DIMSUM+ introduces a new parameter $p$; as a result there are two values to be examined. Since $s$ and $p$ are
independent, two executions with same $\bar{B}_s$ values do not necessarily yield the same result. Therefore, we provide Parameter Finding algorithm as depicted in Algorithm 4 to find the suitable parameter values. Empirically, if the goal is not too close to optimal, we can always find a satisfying pair of $(s, p)$. It is easy to find error convergence over $p$ with a larger $s$ in a very short time\(^3\). Therefore, the inner loop of the algorithm can be replaced with tested results of one or two chosen values. It is worth to address that DIMSUM with $s = 0$ is equivalent to Brute Force in Algorithm 1 while DIMSUM+ with $p \geq L$ is equivalent to DIMSUM.

\(^3\) The evidence is given in the experiment in section 5.3.
Algorithm 4: Parameter Finding \((s, p)\)

for any given \(s\) used by DIMSUM

\[ \text{repeat} \]

\[ \text{reduce } s \text{ slightly} \]

\[ \text{set } p \text{ to a small value} \]

\[ \text{repeat} \]

\[ \text{execute } \text{DIMSUM}^+((s, p)) \]

\[ \text{increase } p \]

\[ \text{until} \]

\[ \text{time increase over accuracy increase rises sharply} \]

\[ \text{if expected accuracy or execution time is met} \]

\[ \text{return } (s, p) \]
CHAPTER 5
EXPERIMENTS

This section shows a comparative study on DIMSUM, DIMSUM+ and Apache Mahout algorithms. We set up experiments, which use both small and large datasets, on a single machine and on clusters, respectively. For the small dataset, we run the algorithms on a laptop computer with 4 CPU cores and 8GB RAM, and for the large dataset, we run the algorithms on Microsoft Azure HDInsight with 9 work nodes of type D3. Each work node has specification of 4 cores, 14GB RAM and 200GB SSD. We evaluate performances of the three algorithms by comparing both accuracy and computational time with different parameter settings. For all of the following experiments involving DIMSUM and DIMSUM+, we set the oversampling factor to $\gamma = 10\log(n) / s$, which is the default value in Apache Spark implementation [10].

There are four experiments to be discussed in the following part of this chapter. Firstly, we find the suitable values of parameter $p$ for the subsequence tests. Secondly, we examine the effectiveness of our algorithm using the parameters found in the previous experiment. Thirdly, we show the linear scalability of our algorithm regarding to an increment of number of dimensions. Finally, we explain the 'missing prediction problem' introduced by Apache Mahout algorithm by demonstrating its impact on
recommendation results.

5.1 Datasets

The MovieLens 1M dataset, referred as the small dataset, is provided by GroupLens project [17]. The dataset contains 1,000,209 ratings given by 6,040 users to 3,900 movies. We split this dataset randomly to 80% training set and 20% test set. The Yahoo! Music dataset, referred as the large dataset, is provided by Yahoo! Webscope Program [18]. This dataset has 717 million rating given by 1.8 million users to 136 thousands of songs. This dataset also comes with both training set and test set. From each user, 10 ratings were chosen randomly into a test set leaving the rest ratings in a training set. Both datasets have ratings in the range of 1 to 5.

5.2 Evaluation Metrics

For performance evaluation with respect to computational time, we focus only on the time spent in calculating similarity of 100 nearest neighbors. The preprocessing and rating prediction times are excluded since they are identical in all algorithms\(^4\). For

\(^4\) The preprocessing takes less than 10 minutes, and prediction usually finishes in 2 minutes for the large dataset.
accuracy evaluation, we predict ratings for all user-item pairs in a test set, and then evaluate the result using Mean Absolute Error (MAE) as specified in (5-1). The lower MAE means more accurate. The root mean square error (RMSE) was also calculated as a plus. Since we observed the same pattern of RMSE and MAE during the tests, the following sections will focus on MAE only.

According to Apache Mahout’s implementation, we discovered the ‘missing prediction problem’ when running it using the large dataset. The detail of such problem is given in section 5.6. As a result, the algorithm is not suitable for the large dataset because it is unable to predict ratings for a large amount of a test set. While making a top \( N \) recommendation, Recall is usually more important than rating accuracy. For services like e-commerce website, there is a limited space to recommend products to customers. A high Recall implies that a large amount of recommendations will be liked/clicked by customers, which leads to more potential transactions. Thus instead of prediction accuracy, we evaluate the Recall for Mahout. We also anticipate that due to such problem, when giving a top \( N \) recommendation, the Recall of Apache Mahout algorithm should be rather low. The equation for Recall is given in (5-2).

It should be noted that for equation (5-1), \( r \) is the real rating from the test set while
prediction is the predicted rating by (1-2), and |testset| is the size of the test set. For equation (5-2), \#hits is the total number of true positives, \(N\) is number of recommendations for each users, and |testusers| is the number of users in the test set.

\[
MAE = \frac{\sum_{r \in testset}|r - prediction|}{|testset|}
\]

(5-1)

\[
Recall = \frac{\#hits}{N \times |testusers|}
\]

(5-2)

5.3 Finding Factor \(p\)

This experiment demonstrates the error convergence of \(p\) for DIMSUM+ on both datasets and finds the suitable values of \(p\) for subsequent experiments. Figure 5-1 shows the surface graph of MAE with various \((p, s)\) values. Since MAE increase/decrease monotonically on both dimensions, it is important to find a pair of \((p, s)\) that yields a

![MAE Surface Graph](image)

Figure 5-1: MAE surface with various \((p, s)\)
close to optimal MAE and requires short runtime. We can also observe from Figure 5-1 that for any value of s, MAE converges quickly when p increases. Thus for this experiment, we start by varying p on both datasets with s = 0.2. For the small dataset, the result shows that the error converges to optimal solution when p ≥ 500 as shown in Figure 5-2. Figure 5-3 depicts the result obtained from the large dataset; it is obvious
that $p = 5,000$ gives a good balance between accuracy and computational time. When increasing $p$ value to 10,000, the computational time increases twice, but MAE decreases by only 0.004. In the following comparative experiments, we choose $p = 500$ for the small dataset and $p = 5000$ for the large dataset.

5.4 Effects of Extra Sampling

To evaluate the effectiveness of our algorithm and to prove our 2nd and 3rd objectives specified in section 1.5, we use the $p$ values found in first experiment for DIMSUM+, cross comparing the runtime and MAE with DIMSUM. We firstly fix $p$ to 500 while varying similarity threshold $s$ in the small dataset; the performance comparison is given in Figure 5-4. With $s < 0.2$, DIMSUM+ consistently gains at least 40% speedup with only $0.2 - 1.1\%$ accuracy loss. When $s = 0$, the comparison becomes DIMSUM+ to Brute Force as specified in Algorithm 1. DIMSUM+ gets 49% speedup with 1.26% accuracy loss. This experiment shows that DIMSUM+ is able to reduce

---

5 The 2nd objective is that the proposed algorithm will produce better accuracy than the original one in a limited time, and the 3rd objective is that the proposed algorithm will require less time than the original one given target accuracy.
computational time efficiently by eliminating the overhead of ‘power users’.

The major objective of DIMSUM+ is to achieve targeted accuracy in shorter time, in other words, to achieve better accuracy in a limited time. To demonstrate the capability of DIMSUM+, we found dozens of suitable values of $p$. We used the procedure provided in Algorithm 3 by varying the values of $s$ in DIMSUM+ from 0.02 to 0.2. We also run DIMSUM by varying the values of $s$ from 0.05 to 0.7. As the

Figure 5-4: Comparing DIMSUM and DIMSUM+ ($p=500$) in the small dataset

Figure 5-5: Prediction quality and runtime for different ($p$, $s$) values
results shown in Figure 5-5, DIMSUM+ provides a huge improvement over DIMSUM. When the targeted MAEs are 0.82 and 0.84, DIMSUM+ is 1.6x and 2x times faster than DIMSUM, respectively. If the response time is the first priority, for example, within one hour, DIMSUM+ is at least 9% more accurate than DIMSUM. We also calculate Root Mean Square Error and observe same results. Finally, the reduction in shuffle size is not as significant as computational time due to the combine function which already minimized shuffle size. We still observe noticeable reduction in shuffle size as shown in Figure 5-6.

![Figure 5-6: Prediction quality and shuffle size for different (p, s) values](image-url)
5.5 Linear Scale with a Growing Number of Users

Apache Mahout algorithm has shown a linear runtime with an increasing number of users [14]. However, the runtime scalability of DIMSUM has not been studied. In this experiment, we examine the scalability of DIMSUM and DIMSUM+. To test the scalability with increasing dimension, we repeated the experiments as specified in [14] using all three algorithms. The Yahoo! Music dataset was split by users into 10 parts, thus it can be used to simulate the increasing number of users by putting more splits in the training set. Figure 5-7 shows the runtime with number of split from 1 to 9 for all algorithms. The 10th split was not included due to its size is significantly smaller than the rest. Since the runtimes are very small, they can be easily affected by many factors such as job scheduling, network or disk I/O, or simply because of the Azure servers’ workload. Therefore, runtime is not a perfect straight line, but we can still see from the

![Graph showing linear scale with number of users with trend lines showing different slopes](image)

Figure 5-7: Linear scale with number of users with trend lines showing different slopes

27
trend lines that all algorithm has a linear increase in runtime. The difference is that DIMSUM has a larger slope compared to Apache Mahout algorithm, and DIMSUM+ is able to reduce that slope to the same level as Apache Mahout. From the equations of trend lines generated by Microsoft Excel, we can intuitively see that DIMSUM has a large slope of 0.76 while DIMSUM+ and Apache Mahout have much smaller slope of 0.4 and 0.3, respectively. We can conclude from the result that DIMSUM+ is more competent in dealing with increasing dimension than DIMSUM.

5.6 Missing Prediction Problem

This experiment explores the impact of ‘missing prediction’ problem on Recall. The Yahoo! Music test set contains at least 10 ratings for each user. For each user, we take the top $N$ items with highest rating as the condition positive. Then we predict ratings for all user-item pairs in the test set, and choose the top $N$ items as recommendation. Therefore, the overlap of condition positive and recommendation is the number of hits, and the size of condition positive is $N \times |testusers|$ as long as $N \leq 10$. We conducted such test with $N = 1$ to 8 for all algorithms with parameters showing in the following table. The test results are given in Figure 5-8.
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Arguments</th>
<th>Time</th>
<th>MAE</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIMSUM</td>
<td>s=0.3</td>
<td>26</td>
<td>0.853165</td>
<td>97%</td>
</tr>
<tr>
<td>DIMSUM+</td>
<td>s=0.1, p=5,000</td>
<td>20</td>
<td>0.835864</td>
<td>97%</td>
</tr>
<tr>
<td>Mahout</td>
<td>p=500</td>
<td>13</td>
<td>0.812335</td>
<td>50%</td>
</tr>
<tr>
<td>Mahout</td>
<td>p=1,000</td>
<td>65</td>
<td>0.802484</td>
<td>60%</td>
</tr>
</tbody>
</table>

Table 5-1: Test parameters

![Graph showing recall for different algorithms using different top N recommendations](image)

Figure 5-8: Recall for algorithms using different top N recommendation

In order to get Table 5-1, we firstly run the Mahout with default parameters (p = 500), then we choose a set of parameters for DIMSUM and DIMSUM+ that has comparable runtime and MAE. The coverage equals to total number of predictions divide by size of test set. While DIMSUM and DIMSUM+ can predict 97% of the ratings, Mahout with default (p=500) settings only has coverage of 50%. We tried to increase p to 1,000, still Mahout only gives 60% coverage. Moreover, the runtime
increased to more than one hour. Thus increasing $p$ to achieve a high coverage is not an option for Mahout. Figure 5-8 shows that DIMSUM and DIMSUM+ have a significantly higher Recall rate for all size of recommendations, especially when $N$ increases, Recall seems to be bounded by coverage. As we mentioned before, increasing $p$ for a better Recall is not a good option for Apache Mahout algorithm. This experiment proved our assumption that in terms of top $N$ recommendations, Apache Mahout algorithm is not suitable despite its low MAE and runtime.
CHAPTER 6
CONCLUSION

This work introduces an enhanced DIMSUM cosine similarity algorithm so called DIMSUM+ using an extra sampling on user vectors. Original DIMSUM algorithm achieves a great improvement in both computational time and network traffic by performing smart sampling over item vectors. The experiments on different size of datasets show that for a given limited time DIMSUM+ is more accurate than the original version, in other words, DIMSUM+ is at least 1.4x faster than the other to obtain the targeted accuracy. Theoretically, DIMSUM+ can be applied to all other similarity measures proposed in DIMSUM. In addition, we also discovered the 'missing prediction problem' for Apache Mahout's implementation, which leads to low Recall rate when making a top N recommendation. Our tests also show that these three algorithms scales linearly with increasing number of dimension, and DIMSUM+ is able to reduce the slope of DIMSUM to the same level of Apache Mahout's implementation.

In the future, we would like to implement and test our algorithm on the latest Apache Spark framework with larger dataset. Also the prediction accuracy of our algorithm can be improved compared to Apache Mahout's implementation.
REFERENCES


Methods with MapReduce," RecSys '12, 9-13 Sept 2012.


APPENDIX A: NOTATIONS

The following symbols are used in this report.

$A$ An input rating matrix

$I$ Set of items from matrix $A$

$U$ Set of users from matrix $A$

$a_{ui}$ User vector for user $u$

$a_{i}$ Item vector for item $i$

$a_{ui}$ Rating given by $u$ to $i$

$S_{ij}$ Similarity value between items $i$ and $j$

$Y$ Oversampling factor for DIMSUM algorithm

$s$ Similarity threshold

$p$ Interaction cut size of Mahout's algorithm, sampling size for DIMSUM+