Collaborative Filtering
Final Report
5/4/16

Tianyi Li, Pranav Nakate, Ziqian Song

Information Storage and Retrieval (CS 5604)
Department of Computer Science
Blacksburg, Virginia – 24061

Dr. Edward A. Fox
Virginia Tech
Executive Summary

The students of CS5604 Information Retrieval and Storage, are building an Information Retrieval System based on tweets and webpages collections of the Digital Library Research Laboratory (DLRL). The students have been grouped into smaller teams such as Front End team, Solr team, Collaborative Filtering team, which are building the individual subsystems of the entire project. The teams are collaborating among themselves to integrate their individual subsystems.

We are the Collaborative Filtering (CF) team, building a recommendation system that can recommend tweets and webpages to users based on content similarity of document pairs as well as user pair similarity. In this final report, we have described the work we have completed on collaborative filtering, including our design for the collaborative filtering system, the algorithms to make content-based and user-based recommendations, the current progress of development and implementation, the lessons we learned, as well as the user manuals for using our system. We have modified the introduction, literature review, project requirement, system design, and future work sections to include our new implementation as well as correcting the format, grammar and spelling errors pointed out in the feedbacks for all the interim reports.

In this final report, we have completed the user-based recommendation implementation in the developer’s manual. The developer manual section is updated from the fourth interim report about the final state of the project. We have discussed our user based recommendation approach in detail in the developer manual section along with the similarity calculation results for content-based recommendations, which we have uploaded into HBase. We have generated fake user data and this fake user information is our starting point to implement the user-based recommendation system with Apache Spark and the Alternating Least Squares (ALS) algorithm.

We have also updated our user manual section. This section describes in detail the steps to be taken to run our code on any new datasets. Finally, we have uploaded our code on a shared drive and have provided a link to it so that anyone can use it.
# Contents

- Executive Summary ........................................................................................................... 1
- List of Figures ..................................................................................................................... 4
- List of Tables ....................................................................................................................... 5
- Introduction ......................................................................................................................... 6
  - Project Requirements ....................................................................................................... 6
    - Content-Based Recommendation .................................................................................. 7
    - User-Based Recommendation ...................................................................................... 7
    - Hybrid Recommendations ............................................................................................ 8
    - Data Requirements ....................................................................................................... 8
- Literature Review .................................................................................................................. 8
- System Design ....................................................................................................................... 11
- Developer Manual ................................................................................................................ 14
  - Download the Webpages .................................................................................................. 14
  - Cosine Similarity Computations for Documents ............................................................ 15
    - Feature Vector Extraction .......................................................................................... 15
    - DIMSUM ..................................................................................................................... 17
    - Exporting Similar Documents ..................................................................................... 18
  - Alternating least squares for user-based recommendation ............................................. 19
    - Randomly generated fake user data .......................................................................... 19
    - Latent factor model ...................................................................................................... 20
    - Matrix factorization methods ..................................................................................... 20
    - Collaborative Filtering - spark.mllib ......................................................................... 21
  - Data input and output ........................................................................................................ 21
- User Manual .......................................................................................................................... 22
  - User manual for content-based recommendations .......................................................... 22
    - Build Process ............................................................................................................... 22
    - Moving the JAR file ...................................................................................................... 23
    - Submitting the Spark job ............................................................................................. 23
    - Processing the output files ......................................................................................... 23
    - Upload to HBase .......................................................................................................... 24
    - Batch processing ......................................................................................................... 24
  - User manual for user-based recommendations ................................................................ 25
List of Figures

Figure 1 Basic idea of CF................................................................. 11
Figure 2 Item-based recommendation system ........................................... 12
Figure 3 A variant of content-based recommendation system........................... 12
Figure 4 Collaborative filtering viewed as an m\*n matrix with partially filled ratings, representing user judgments ................................................................. 13
Figure 5 Files generated using the script ....................................................... 14
Figure 6 Screenshot of short URLs we extracted ............................................ 15
Figure 7 Screenshot of long URLs we extracted ............................................. 15
Figure 8 Document Similarity system pipeline .............................................. 15
Figure 9 Document Feature Vectors Matrix .................................................. 17
Figure 10 HBase column family ................................................................... 18
Figure 11 HBase columns after export step ..................................................... 19
Figure 12 Recommendation data flow ............................................................ 21
Figure 13 Directory structure for build process ............................................. 22
Figure 15 Directory structure for user recommendation build process .......... 25
Figure 16 Screenshot of final output for user recommendation ....................... 26
Figure 17 Tweet input file ........................................................................... 27
Figure 18 List of tweets each term appears in; output of first MapReduce job .... 28
Figure 19 List of tweet IDs similar to each tweet; output of second MapReduce job .... 28
Figure 20 Compute similarity between tweets ............................................... 28
Figure 21 Tfidf score of each term in different tweets ..................................... 28
Figure 22 First nine tweets in the cluster, for testing purpose ......................... 29
Figure 23 Tweets similarity matrix (of first nine tweets) ................................ 29
Figure 24 Pipline of execution ..................................................................... 30
Figure 25 output of mapper of MR-1 ............................................................. 31
Figure 26 Output of mapper of MR-2 ............................................................ 32
Figure 27 Output of reducer of MR-2 ........................................................... 32
List of Tables

Table 1 Inventory of all the source files ................................................................. 22
Table 2 Tentative timeline of the project .................................................................. 36
1 Introduction

Collaborative filtering, commonly called recommendation, is often built into advanced information retrieval systems and recommends items based on current search results. Despite the name, which is usually used to refer specifically to user-based recommendation, our project will make use of a hybrid of both content-based and user-based approaches to recommend related documents to each query search.

There are three major filtering algorithms to make recommendations. Content-based filtering is to learn what kinds of contents a user likes and then match the contents of a current article with a “content prototype” that we believe describes well what the user likes [3]. Collaborative filtering (user-based filtering) assumes that if users who are similar to the current user like some items, the current user might also like it [3]. A hybrid recommender combines the two, probably also involving knowledge-based and demographic techniques.

The goal of our recommendation system is to better assist users in finding the information they are looking for without having to enter a very concise query to the information retrieval system. For this project, we will be reading in the user query and the corresponding results from the Solr team, along with all the user query history available, to do some back-end recommending computation that leads to a list of related documents, so that the Front End team will be able to show recommendations of items that might also be of interest to the users.

In this final report, we first introduce the background and goals of our team, offering a high-level overview about where our part fits in the whole course project and why it is important. Then we describe both the requirements set for our team and what our data requirements from other teams, setting out tasks and expectations. After that, in the literature review part, we discussed recommendation systems research in general, the status quo and models and algorithms that we will use for this project. Following that, we presented our current system design, talked about the challenges we face, the solutions we have come up with, and how we plan to build the system. Subsequently, we discussed the evaluation metrics to ensure the quality of the results as well as the performance of our recommendation system. Finally, we laid out our plans and milestones in a semester timeline table along with our current progress.

2 Project Requirements

The end goal of our team is to build a recommendation system which can recommend tweets and webpages to a user who searches for documents of either type within a particular context.
In the beginning, we don’t have real user data; the system will recommend tweets and webpages to the users based on the content similarity. When we have users clicking our recommendation results, we will calculate the user similarity based on their clicking history. With that, we will continue to recommend tweets and webpages according to both content and user similarity.

2.1 CONTENT-BASED RECOMMENDATION

In the initial phase of the system, we will not have enough user data. Therefore, we will use item-based recommendation in the early phase.

We identify the following requirements for this type of recommendation:

1. Extract the document features and create document vectors for similarity metric calculations
2. Create a document-document similarity table, by computing cosine similarity for each item pair
3. Filter the top K similar documents for a document
4. Order the recommended documents based on their cosine similarity score with each document.

The tweets and webpages our system recommends, may not contain the user input from the original search query, so they won’t be ranked on the top of search results. But our recommendation system will show these similar documents to the user if they are interested by clicking on the ‘see more’ anchor link given for each search result.

2.2 USER-BASED RECOMMENDATION

Once users start using the information retrieval system built by the CS5604 IR class, we expect the system will accumulate large user browsing history wherein each user would have viewed on average 10 documents such as a tweet or a webpage. We can assume that the browsing history of a user reflects the user’s past interests. In this type of recommendation, we intend to learn the user behavior patterns by comparing a user with other users. By looking at the user’s search and browsing history, we can model the user’s context, preferences and interests. Hence based on the user’s search and browsing history, we intend to build a personalized document recommendation system. For this type of recommendation, we identify the following requirements:

1. Extract user browsing history features for similarity metric calculations
2. Create a user-user similarity table, by computing the user-user similarity metric for each user-pair
3. Find and rank the documents viewed by top K similar users
4. Order the documents per their ranks and output them as recommendations
If we can get user clickstream data for documents from either the Solr team or Front End team, we will be able to build a weighted array of potentially related documents. When another user clicks on one of the documents in the list, we can find similar documents within this list first, giving the results more weight than those from the regular approach. For now, we still don’t have the user data. So we generated random user data and perform user recommendation based on the fake user data.

2.3 HYBRID RECOMMENDATIONS
After the system accumulates more user data, such as user queries, and viewing or clicking history of tweets and webpages by each user, our system will recommend tweets and/or webpages to the users based on a hybrid approach of user-user similarity and item-item similarity. We generated some faked random user data to get started, and will tailor our system according to real data once the Front End team figures out what user data format they will be able to provide us.

2.4 DATA REQUIREMENTS
Following are our data requirements for the Solr team and Front End team:

1. User names or IDs and the association of each user with the following activities.
2. Search queries issued by these users.
3. Document browsing history of users, such as tweets and/or webpages clicked on (and viewed) by these users.

The Front End and Solr teams together are fulfilling the first 2 data requirements. And together these 2 teams are working on our third data requirement.

3 Literature Review
Many studies of the existing Recommendation systems literature use Collaborative Filtering (CF) on similarity measures between users and/or items. In the Collaborative Filtering approach, the items recommended to a user are those that similar users already like, or similar items which have high similarity to the items that the user is known to like.

In [3], Collaborative Filtering is defined purely in terms of user-user similarity measures. The item-based filtering is defined as a separate branch from collaborative filtering. Item based filtering is to learn the kind of content a user likes. To this user the system recommends the item that has high similarity with a item prototype that the user likes. The idea is to inspect the content of the item in the global item set and compare it to the active user's preferences without any consideration of other users and their similarity to the active user.

This work also gives an abstract overview of a content based filtering system. The system consists mainly of three components. The first is the initialization module
which gets the system started by reading in the user profile. In case of document recommendations, the user profile can be a short summary of the user preferences such as the general reading habits of the user, the areas of interest, etc. The second component is the decision module which acts as a binary classifier. Based on the user profile this classifier determines whether to deliver a particular item (document) to the user or not. The third is the feedback module or the learning module. If the user happens to give a feedback on the delivered document, the learning module can signal the classifier and update its parameters. The user may not give an explicit feedback; however, we can derive implicit feedback from the user based on whether the user viewed the document or not.

In the context of this project we will implement a similar system. We believe we can use a different form of decision module from the system described above. Since we will not have any user data initially, we will be facing the widely popular cold start problem. We would have to rely on item-item filtering to recommend documents to the user when the user issues search queries. For the given data set, we plan to build a document-to-document similarity matrix by iterating through all document pairs and computing a similarity metric for each pair. This process will incur $O(n^2)$ time, which is a computationally expensive task. However, this will be a one-time computation. Every time a new document is added to the set, we will calculate only the similarity metric of this document with other documents incurring $O(n)$ time. For a search result, we will look up its top $K$ similar documents and recommend them to the user. The learning module will extract the feedback from the user behavior towards every document and it will update the similarity measures of these recommended documents.

In summary, this will be our initial approach for the project, based on document-to-document collaborative filtering. The key to the scalability and performance of this approach is that it creates the document-document similarity table offline. The online component looks up similar documents for the user's preferences of documents, aggregates the scores and displays them as recommendations. This approach will scale irrespective of the user base and it will only depend on the documents displayed in the search results. Since the system will recommend only the highly correlated items, we expect the quality of recommendations to be high.

This work also describes the pure collaborative filtering based on user-user similarity measure, which make two basic assumptions. The first is that the users with a common interest will have similar item preferences and second that the users with similar preferences share the same interest. For example, two friends with similar preferences for books are very likely to read what the other has already read. In this approach we infer an individual's interest based on other similar users. The actual content of the items these similar users prefer does not have any significance. However, this approach requires a large amount of user preferences data to be available, which is not the case in the context of the current project. We have requested other teams to provide us with the user preferences
data, and once we have that we will be able to make user-based recommendations, which tend to be more accurate in real-world applications.

The Amazon.com recommendation system as described in [1] also employs several interesting approaches. It identifies pure user-user collaborative filtering as computationally expensive incurring $O(MN)$ in the worst case, where $M$ is the number of distinct users and $N$ is the number of product catalog items. The work mentions the scaling issues that are encountered in this approach and advises using dimensionality reduction by partitioning product categories or subject classification. Dimensionality reduction can reduce $M$ or $N$ by a large factor. However, the paper admits that with this approach the quality of recommendations is low, since the space of user and/or items reduces for matching user's preferences.

The Amazon.com recommendation system [1] also gives a new clustering approach towards recommendations. To find users who are similar to the active user being considered, clustering divides the user space into many clusters. The goal is to assign the user to the segment containing the most similar customers. It then uses the cluster users as the base to calculate user-user similarity measures and make recommendations based on similar users' preferences. Clustering the global user base comes under unsupervised learning and it can be done in offline computation mode. The cluster models have better online scalability and performance than pure collaborative filtering since they compare the user to a controlled number of segments rather than the entire user base. However, the paper highlights the same flaw as before, the quality of recommendations is low. This is because the similar customers that the clustering algorithms find may not be the most similar customers and hence the recommendations can be less relevant. The quality can be improved by using numerous fine-grained segments, but then it is just as expensive as pure user-user collaborative filtering.

Similar to document recommendations as required in this project, we looked at recommendation systems in other domains such as music recommendation. The Yahoo! Music Recommendation system [4] describes a Matrix factorization model, which is a variant of CF approach, to exploit the information of their own dataset. The basic idea in this approach as with many others is that, a matrix of users is created where each row is a user. Every column of this matrix represents a particular item. For example, if $M$ is the user-item matrix then $M_{i,j}$ represents the numerical rating or binary feedback by user $i$ for the item $j$. In matrix factorization, this matrix is decomposed to create two matrices, $X_{k,i}$ and $Y_{k,j}$, such that $M = X^T \times Y$. In doing so, we get an embedding of both users and items into a smaller dimensional space. Aioli [1] mentioned that Matrix factorization, which comes under Model-based Collaborative filtering, is computationally intensive for large datasets. Instead, memory-based Collaborative Filtering used to compute the recommendations using a user-item matrix and the scoring functions for
determining ranking were computed on an item for each user using an appropriate weight to make music recommendations.

After going over all this previous work, we initially thought that we need an approach to reduce the number of document comparisons while finding similar documents for each document. In the period until Interim Report 2, we came up with a system design that involves two MapReduce stages in the execution pipeline described in the implementation section. The output of the second MapReduce stage would give us the top K similar documents for each document using the Hadoop MapReduce.

After perusing the internet for a solution and consulting with the GRAs, we came across an efficient implementation [6] to calculate cosine similarity of each vector with other vectors, in a large collection of vectors. This implementation of the DIMSUM algorithm is part of the Apache Spark’s [5] machine learning library. Hence we have now come up with a system design, based on Apache Spark’s MLlib packages. We describe this design in the next few sections.

4 System Design

The corpus in which our recommendations happen have six categories. Each has two sub-categories of tweets collections. This dataset serves as a test bed in the testing phase of our recommendation system, using content-based item-to-item algorithms when there is no prior user query and browsing history available. Figure 1 shows the basic idea of collaborative filtering.

In user-based collaborative filtering, as shown in the left side of Figure 1, we make recommendations by finding similar users for an active user. The third user in the Figure 1 has high similarity with the first user and then the second user. Hence we can recommend the items consumed by user 1 and user 2 to user 3. For example, in Figure 1, item 1 and tweet 4 are recommended to user 3.
In item-based collaborative filtering, shown in the right side of Figure 1, similarity between items is used as the basis of recommendations. For tweet 1 and tweet 3, similarity is high since those items have intersecting user sets (two common users, user 1 and user 2). Hence we can recommend tweet 1 to user 3 who has consumed tweet 3 but not tweet 1. Also, item 1 and tweet 4 have an intersecting user, user 1. And hence item 4 can also be recommended to user 3 but the likelihood of user 3 liking tweet 4 is less, since its similarity to tweet 3, which is already consumed by user 3, is less (one common user, user 1).

The item-based recommendation system that we intend to implement in the initial phase, is shown in the Figures 2 and 3.

Figure 2 Item-based recommendation system

Figure 3 A variant of content-based recommendation system

Figure 2 and Figure 3 show our content-based recommendation system. As shown in Figure 2, we will filter each document from the corpus, based on its content similarity metric value with the document from search results that the user gets.
Figure 3 shows the variant of the earlier system that we will implement. In this approach we assume to have user history in the system. Based on the browsing history of a user, the system will learn the preferences and interests of the user. A binary classifier will read this information about the user’s interests. The classifier will inspect the content of the tweets and webpages and compare it to both the user’s preferences and categorize them as relevant and non-relevant. The user may give feedback on these recommendations or we can derive implicit feedback based on whether the user browses the recommended document. The feedback will be given to the learning module, which will update the classifier parameters. This system will make recommendations without considering information from other users.

Figure 4 extends our model into a user-item matrix, where each entry is the rating of a user on a certain tweet/webpage. Each row can be viewed as a user vector, and each column an item vector. The item-based algorithm discussed above can be considered as computing the cosine similarity of the item vectors; whereas the collaborative filtering takes each user vector and seeks to find similar users to the current one.

![Tweets/Webpages: T](image)

<table>
<thead>
<tr>
<th>Users: U</th>
<th>T1</th>
<th>T2</th>
<th>...</th>
<th>Ti</th>
<th>...</th>
<th>Tn</th>
</tr>
</thead>
<tbody>
<tr>
<td>U1</td>
<td>R11</td>
<td>R12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U2</td>
<td>R21</td>
<td>R22</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ui</td>
<td>R11</td>
<td>R12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Um</td>
<td>Rm1</td>
<td>Rm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Ratings: $R = f(Ui, Ti)$

$f: U^*T \rightarrow R$

*Figure 4 Collaborative filtering viewed as an $m^*n$ matrix with partially filled ratings, representing user judgments*

Our recommendation system will be integrated with Apache Solr and the Hue interface of the Front End team. We will package our recommender system, which consist of Scala scripts to be run on Apache Spark, in zip files along with
the SBT build script [7]. The system will take the collection folder paths, user browsing history logs as its input, and upload the results of item-based and user-based recommendations in appropriate column families in HBase.

5 Developer Manual

We begin our implementation with calculating the similarity between tweets. In order to calculate the similarity, we have to have the cleaned tweets. In the beginning, the collection management team had not finished their cleaning, so we’ve done a simple cleaning job by ourselves. We removed stop words and profane words in addition to non-English words, symbols and URLs in the tweets. The process also involves word lemmatization, which is required since we rely on a document-document similarity in the initial phase.

5.1 Download the Webpages

We will also calculate the document similarity between webpages. In order to get started with development, we use a script “downloadWebpagesFromTweets.py” to download all the articles in tweets from the web. But after the collection management team provided us with cleaned webpages, we are using the webpages provided by them. The script extracts the URLs of all the tweets. Then it will translate URLs in abbreviation version to their complete version and give out a sorted result for the URLs in the tweets file. We are using part of the data from tweets collection z_541 for testing, which contains 282 tweets. Figure 5 shows the files that we have generated using the script. It includes all the short URLs in “z541Test-ShortURLs.txt”, and long URLs in “z541Test-LongURLs.txt”. The other .txt files are webpages we downloaded using the script.

![Figure 5 Files generated using the script](image)

By running the script, we get short URLs list from the tweets; the result shows in the .txt file “z541Test-ShortURLs.txt”. Figure 6 is a screenshot of the short URLs we extracted.
We expand the short URLs to long URLs, then we download them in our local machine. The long URL list shows in the file .txt file “z541Test-ShortURLs.txt”. Figure 7 is a screenshot of long URLs we extracted.

5.2 COSINE SIMILARITY COMPUTATIONS FOR DOCUMENTS

We are using the DIMSUM algorithm to do item-based recommendations, which is implemented as a method in Apache Spark’s machine learning library MLlib. Our system pipeline for the item-based recommendations is as shown in figure 8.

5.2.1 Feature Vector Extraction

In the first step, we convert all the documents to their feature vector form.
Term frequency-inverse document frequency (TF-IDF) is a feature vectorization method widely used in text mining to reflect the importance of a term to a document in the corpus. Denote a term by $t$, a document by $d$, and the corpus by $D$. Term frequency $TF(t,d)$ is the number of times that term $t$ appears in document $d$, while document frequency $DF(t,D)$ is the number of documents that contains term $t$. If we only use term frequency to measure the importance, it is very easy to over-emphasize terms that appear very often but carry little information about the document, e.g., “a”, “the”, and “of”. If a term appears very often across the corpus, it means it doesn’t carry special information about a particular document. Inverse document frequency is a numerical measure of how much information a term provides:

$$IDF(t,D) = \log \frac{|D| + 1}{DF(t,D) + 1}$$

where $|D|$ is the total number of documents in the corpus. Since the logarithm is used, if a term appears in all documents, its IDF value becomes 0. Note that a smoothing term is applied to avoid dividing by zero for terms outside the corpus. The TF-IDF measure is simply the product of TF and IDF:

$$TFIDF(t,d,D) = TF(t,d) \cdot IDF(t,D)$$

The tf-idf weights are computed for each term of each document by the spark module. Using these weights, we created a feature vector for each document. Each such document vector is then inserted as a row in the document feature vectors matrix (let’s call it A), which is shown in Figure 9.

We use HashingTF library of Apache Spark’s MLlib package. This algorithm hashes each term of the document over a large but limited range of hash buckets. The term frequencies are calculated by counting the terms that map to the same hash bucket. Chances of collision i.e. two different terms being mapped to the same bucket are highly unlikely if the target feature dimension which is the number of hash buckets is very large. The default value of the feature dimension is $2^{20} = 1,048,576$. The IDF is calculated based on the statistics generated by the same hashing technique and hence each document essentially has $2^{20}$ features most of which are 0, since only the terms actually present in a given document will have non-zero tf-idf weight. The spark MLlib creates a dense vector for each document by eliminating the terms with 0 tf-idf weight.
Once we have the document feature vectors matrix ($A$), we use the DIMSUM algorithm [6], to calculate the cosine similarity between all pairs of the rows of $A$. The algorithm is implemented as a method in the MLlib package of Apache Spark. The algorithm effectively calculates the following:

$$A \cdot AT$$

From the documentation, we found out that the algorithm first computes the dot product of the matrices and then normalizes them with the magnitude of the respective vectors. Since each row of $A$ is one document vector, each column of $AT$ is a document vector as well. Hence their dot product is essentially the cosine similarity of the document vectors.

From the documentation, we also found out a very important property of DIMSUM. The algorithm computes column similarity for each pair of column similarity. However, our document vectors are arranged in the document feature matrix row-wise meaning one document vector is one row of the document feature matrix. Therefore, to apply the DIMSUM algorithm on our input document feature matrix, we will take transpose it first.

This algorithm computes similarities between columns of this matrix using a sampling approach. It takes one input which is the threshold parameter. The threshold parameter is a trade-off knob between estimate quality and computational cost.

Setting a threshold of 0 guarantees deterministic correct results, but comes at the same cost as the brute-force approach. Setting the threshold to positive values incurs strictly less computational cost than the brute-force approach, however the similarities computed will be estimates. The sampling guarantees relative-error correctness for those pairs of columns that have similarity greater than the given similarity threshold.
The DIMSUM algorithm parallelizes the matrix multiplication along with other optimization techniques. The most understandable optimization is that, it only computes the upper diagonal matrix of the product, since similarity of vector $i$ with vector $j$ is the same as the similarity of vector $j$ with vector $i$.

The second optimization technique helps parallelization when the underlying resilient distributed dataset (RDD) of the document feature matrix is partitioned into multiple chunks. This allows multiple executors of the cluster to do the computations in parallel and hence speed up the execution. We observed that repartitioning can help us to compute document similarities for large number of documents in one spark job.

### 5.2.3 Exporting Similar Documents

Once we get the cosine similarity between all pairs of document vectors, we sort the similar documents for each document, in the descending order of their similarity with the current document. We then export these similar documents to HBase column family as shown in Figure 10.

<table>
<thead>
<tr>
<th>Rowkey (ID of the document)</th>
<th>sim_doc_id (ID of similar documents, in .csv)</th>
<th>sim_score (Similarity score of documents, same order, in .csv)</th>
<th>URL1</th>
<th>URL2</th>
<th>URL3</th>
</tr>
</thead>
<tbody>
<tr>
<td>TweetID/WebpageID</td>
<td>tweetID1, tweetID2, tweetID3</td>
<td>simScore1, simScore2, simScore3</td>
<td>tweet_URL1</td>
<td>tweet_URL2</td>
<td>tweet_URL3</td>
</tr>
</tbody>
</table>

*Figure 10 HBase column family*

Figure 10 shows our table in HBase. The table contains two columns. The Rowkey contains the IDs of tweets or webpages. The first column “sim_doc_id” stores the IDs of similar tweets corresponding to the IDs in the Rowkey in CSV format. We store the similar tweets in descending order based on the similarity score. The second column stores the similarity score in CSV format. The other three columns store the URLs of the top 3 similar tweets. They are required by the front-end team to show our results.

After we executed our scripts on a small tweet collection, our column family in HBase has data as shown following:
Figure 11 HBase columns after export step

<table>
<thead>
<tr>
<th>Rowkey (ID of the document)</th>
<th>sim_doc_id (ID of similar documents, in .csv)</th>
<th>sim_score (Similarity score of documents, same order, in .csv)</th>
<th>URL1</th>
<th>URL2</th>
<th>URL3</th>
</tr>
</thead>
<tbody>
<tr>
<td>541-553713617707626496</td>
<td>541-555079653954957312, 541-55323869132628832, 541-55586344187031552</td>
<td>0.639485069837304950, 0.633024189436729123, 0.20526364985236712</td>
<td><a href="http://t.co/05WntOXEx2">http://t.co/05WntOXEx2</a></td>
<td><a href="http://t.co/ijrKLKSM">http://t.co/ijrKLKSM</a></td>
<td><a href="http://t.co/NZPPU3jg6R">http://t.co/NZPPU3jg6R</a></td>
</tr>
<tr>
<td>541-55405327073349633</td>
<td>541-555133548965801895, 541-553708266413654016, 541-5547633249836311041</td>
<td>0.791304958684930405, 0.776472839463215983, 0.773458604929203058</td>
<td><a href="http://t.co/VTewqEsiQ">http://t.co/VTewqEsiQ</a></td>
<td><a href="http://t.co/QnkTsTv16le">http://t.co/QnkTsTv16le</a></td>
<td><a href="http://t.co/18aVeIPYUB">http://t.co/18aVeIPYUB</a></td>
</tr>
<tr>
<td><a href="http://t.co/VTzwqERsp">http://t.co/VTzwqERsp</a></td>
<td><a href="http://t.co/w7WntEXUx2">http://t.co/w7WntEXUx2</a>, <a href="http://t.co/FrYiy2ZMK4M">http://t.co/FrYiy2ZMK4M</a>, <a href="http://t.co/VZQQS1jg6">http://t.co/VZQQS1jg6</a></td>
<td>0.856749958473947592, 0.823058478395068594, 0.773405694850639485</td>
<td><a href="http://t.co/17ywYSu42U">http://t.co/17ywYSu42U</a></td>
<td><a href="http://t.co/FGbJwi67t">http://t.co/FGbJwi67t</a></td>
<td><a href="http://t.co/1yaABeEigP">http://t.co/1yaABeEigP</a></td>
</tr>
<tr>
<td><a href="http://t.co/QnksTv16le">http://t.co/QnksTv16le</a></td>
<td><a href="http://t.co/01yQvD0a42l">http://t.co/01yQvD0a42l</a>, <a href="http://t.co/Chbhw0F4Jt">http://t.co/Chbhw0F4Jt</a>, <a href="http://t.co/1yaBw5VgB">http://t.co/1yaBw5VgB</a></td>
<td>0.763494050485950928, 0.739405059404930409, 0.638495960031394054</td>
<td><a href="http://t.co/5YaRru5TbgD">http://t.co/5YaRru5TbgD</a></td>
<td><a href="http://t.co/1yBxe5VgN">http://t.co/1yBxe5VgN</a></td>
<td><a href="http://t.co/9oaNAeEkgl">http://t.co/9oaNAeEkgl</a></td>
</tr>
</tbody>
</table>

Figure 11 shows an example. We store all the similar tweets in the column “sim_doc_id”. We store the similarity score in the column “sim_score”. We store the URLs of the top 3 similar tweets in the other three columns for the front-end team to make use of our recommendation results. For tweet with ID “541-553713617707626496”, the tweet with ID “541-555079653954957312” has the highest similarity score. Tweet ID “541-555079653954957312” is stored in the column “sim_doc_id”, on the top of the CSV file. The score is “0.63948506983730495”, which is stored in column “sim_score”, on the top of the CSV file. For webpages with ID “http://t.co/VTzwqERsp”, the webpage with ID http://t.co/w7WntEXUx2 has the highest score. So, the ID is stored on the top of the CSV in column “sim_doc_id”. The score is “0.85674995847394759”, stored in column “sim_score”.

5.3 ALTERNATING LEAST SQUARES FOR USER-BASED RECOMMENDATION

User-based recommendation, also known as collaborative filtering, analyzes relationship between users and interdependencies among products to identify new user-item associations [8]. One advantage of this approach over content-based recommendation is that it is domain free yet addresses the aspects of documents that are elusive and difficult to profile by looking at static contents only.

5.3.1 Randomly generated fake user data

While generally more accurate than content-based techniques, user-based recommendation suffers from the cold start problem, since we do not have any users in the IDEAL system at the very beginning. This is why we started with document similarity computation; as more users coming in, we will have their
clickstreams and other implicit feedbacks on the documents, with which we can add user-based techniques in our recommendation system. For development purpose, we randomly generated a number of user logs to bypass the cold start problem and set up the user-based recommendation system. Following are two assumptions we made about the potential user data and are subject to change according to future real data:

- User ID is an integer value ranging from 1 to N, where N is the total number of users (increasing with more user coming in)
- The number of users is more than the number of documents viewed. In real world, we will come to this point and this is also when the user-based recommendation starts to have stronger impact than content-based recommendation.

User IDs and document IDs are generated by random integer generator from Python random package. Please see complete code in appendix.

The output of our user data, which is also the input for our user-recommendation system is shown in Figure 12. Each row has three field (user_id, document_id, rating_score), which tells if this user has viewed this document or not, see Figure 12.

5.3.2 Latent factor model
Latent factor models are an alternative to the traditional neighborhood methods. It tries to explain the ratings by characterizing both documents and users on a number of factors inferred from the rating patterns. In our case, since the user information (logs) that will be available to us is not decided yet, we will start with one single factor that whether the user has clicked on a document or not. If we can get more detailed user logs, for example, the clickstreams including how many times the user clicked the document, the amount of time the user spent on a document, etc. we will continue to work on more complex methods to model those data and compute more informative ratings.

5.3.3 Matrix factorization methods
Matrix factorization methods are one of the most common realizations of latent factor models, especially when explicit feedbacks from users are not available. It characterizes both users and documents by factors of factors inferred from document rating patterns. In our case, we use binary ratings to represent if the user has viewed (clicked on) a document (rating score = 1) or not (rating score = 0). With this, we do not need to do complicated matrix factorization. However, in order to make our implementation more extendable with potential user information, we still map both users and documents to a joint latent factors space of dimensionality $f$, such that user-item interactions are modeled as inner products in the space. Accordingly, each document $d$ is associated with a vector $V_d \in \mathbb{R}^f$, each user $V_u \in \mathbb{R}^f$, and the approximated rating of user $u$ of document $d$ is denoted by $R_{u,d}$, which is the dot product of the two vectors: $R_{u,d} = V_d \cdot V_u$. 
5.3.4 Collaborative Filtering - spark.mllib

We made use of the spark.mllib collaborative filtering package to compute the mapping of documents and users to factor vectors, which implemented the ALS (Alternating least squares) learning algorithm. In this section, we focus on explaining our use of the package and omit the algorithmic details, readers can read more about the algorithm in referenced paper.

The ALS package allows us to train a recommendation model by providing initial training data set and specify details like the number of blocks used to parallelize computation, the number of latent factors in the model, the number of iterations to run and so on. For fake user data, we are using 10 blocks to parallelize computation for 10 iterations, setting baseline confidence to be 0.01. Given the trained model, we are recommending the top 3 documents to all users.

And we are ready to tailor our system to recommend a specified number of documents to all users or a particular user with the provided API, according to what suits the Front-end team better. Complete code can be found in appendix.

5.4 DATE INPUT AND OUTPUT

Figure 12 shows the input and output data of both recommendation systems. For user recommendation, we have three data input: userID; tweetID or webpageID; clicked or not clicked, indicated by 1/0. The output of our recommender is userID, IDs of similar tweets or webpages, and corresponding similarity score.

For item recommendation, our input is tweetID or webpageID and the content of the tweets or webpages. Our output is tweetID or webpageID, IDs of similar tweets or webpages, and corresponding similarity score.

![Diagram of recommendation data flow]

*Figure 12 Recommendation data flow*
### Table 1 Inventory of all the source files

<table>
<thead>
<tr>
<th>Item-based Recommendation</th>
<th>Filename</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DocumentSimilarity.scala</td>
<td>Compute similarity for each document pair</td>
</tr>
<tr>
<td></td>
<td>simple.sbt</td>
<td>Compile and build the Scala script for document similarity</td>
</tr>
<tr>
<td></td>
<td>ParseResult.py</td>
<td>Parse the result part files</td>
</tr>
<tr>
<td></td>
<td>BatchProcess.py</td>
<td>Batch process the result part files</td>
</tr>
<tr>
<td>User-based Recommendation</td>
<td>NewUserSim.scala</td>
<td>Generate document lists based on user interest</td>
</tr>
<tr>
<td></td>
<td>simple.sbt</td>
<td>Compile and build the Scala script for user recommendation</td>
</tr>
<tr>
<td></td>
<td>parseData.py</td>
<td>Parse the result files</td>
</tr>
<tr>
<td></td>
<td>generateRandomUser.py</td>
<td>Generate random users to avoid cold-start problem</td>
</tr>
</tbody>
</table>

### 6 User Manual

#### 6.1 USER MANUAL FOR CONTENT-BASED RECOMMENDATIONS

#### 6.1.1 Build Process

We have created a sbt script to build our Scala code. This script will compile all the scala scripts into class files and package them together in a java archive file (JAR). This is essential when we submit a Spark job using the `spark-submit`. To build our Scala code, please make sure you have the two files namely the ‘DocumentSimilarity.scala’ and ‘simple.sbt’ in the same folder, typically a subfolder in your home directory.

We have a folder named ‘Document Similarity’ in which we placed these two files. Our build setup is shown in the next figure.

![Figure 13 Directory structure for build process](image-url)
We issue the following command on the terminal as shown in the figure above:


`sbt package`

This creates the jar file in the ‘target’ folder that we will use to submit a Spark job. The ‘project’ and ‘target’ folder are created by the sbt tool.

6.1.2 Moving the JAR file

Next we move the jar file created in the step 1 to the home folder, from where we will submit the Spark job. To do this issue these 2 commands on the terminal:

```
cd
mv DocumentSimilarity/target/scala-2.10/document-similarity_2.10-1.0.jar /home/cs5604s16_cf/
```

To go back to the home folder. To move the jar file to the home folder. Please note that it is not necessary for the jar file to be in your home folder, but to standardize the Spark job submission process and to process the results it would be convenient to work from the home folder.

6.1.3 Submitting the Spark job

To submit the Spark job to from the home folder issue the following from the terminal:

```
spark-submit --master spark://node1.dlrl:7077 --class DocumentSimilarity --jars document-similarity_2.10-1.0.jar document-similarity_2.10-1.0.jar /user/cs5604s16_cf/test2 /user/cs5604s16_cf/cluster_10k 0.2
```

The parameters needed by the script are highlighted by red, blue and yellow color in the above command.
The first parameter (in green) is the path of the input file on HDFS.
The second parameter (in red) is the path of the folder on HDFS where the output files will be written by the Spark job.
The third parameter (in yellow) is the threshold used by the similarity computation implementation of the Spark library.

6.1.4 Processing the output files

To upload the results on the HBase column family, we need some minor processing of the results to comply with the input file format needed by the HBase TSV file import module.
First we download the files from HDFS to the home folder with the following command:

```
hadoop fs -get /user/cs5604s16_cf/sim_10k/part-00000 ./part1
```

This must be done for all the output files in the output folder specified in step 3 above.

Then the file must be parsed line by line by the script (ParseResults.py) to make it comply with the TSV format needed by HBase file importer.

```
./ParseResult.py part1
```

This script will create a new file with the same name but with ‘_new’ appended at the end.

### 6.1.5 Upload to HBase

Once we process all the output files, we can upload them to HBase using the following command:

```
```

The red colored text are the two column families where our data is being uploaded. The blue colored text is the name of the table name in HBase. The yellow colored text is the processed output file.

The format of the line in the processed output file is as below:

```
ROWKEY sim_tweet_1, sim_tweet_2, sim_tweet_3 score_1, score_2, score_3
```

Note that the fields in the line are tab separated.

### 6.1.6 Batch processing

We have also created a python script that will automate the last 2 steps. The name of the script is ‘BatchProcess.py’. This script will take the path of the HDFS output folder of the Spark job. It will parse all the files in that folder and upload the results in HBase. This automation is necessary if you partition the data into a large number of chunks in which case the number of output part files generated is equal to the number of partitions/chunks.

```
./BatchProcess.py path-to-hdfs-output-folder
```
6.2 USER MANUAL FOR USER-BASED RECOMMENDATIONS

6.2.1 Build Process

We have created a sbt script to build our Scala code. To build the given Scala code, please make sure you have the two files namely the ‘NewUserSim.scala’ and ‘simple.sbt’ in the same folder, typically a subfolder in your home directory. We have a folder named ‘userSimilarity’ in which we placed these two files. Our build setup is shown in the next figure.

![Directory structure for user recommendation build process](image)

We issue the following command on the terminal as shown in the figure above:

```
sbt package
```

This creates the jar file in the ‘target’ folder that we will use to submit a spark job. The ‘project’ and ‘target’ folder are created by the sbt tool.

6.2.2 Moving the JAR file

Next we move the jar file created in previous step to the home folder, from where we will submit the spark job. To do this issue these 2 commands on the terminal:

```
cd
mv userSimilarity/target/scala-2.10/user-similarity_2.10-1.0.jar /home/cs5604s16_cf/
```

To move the jar file to the home folder. Please note that it is not necessary for the jar file to be in your home folder, but to standardize the spark job submission process and to process the results it would be convenient to work from the home folder.

6.2.3 Submitting the Spark job

To submit the spark job to from the home folder issue the following from the terminal:

```
spark-submit --master spark://node1.dhrl:7077 --class UserSimilarity --jars document-similarity_2.10-1.0.jar document-similarity_2.10-1.0.jar /user/cs5604s16_cf/test2 /user/cs5604s16_cf/cluster_10k 0.2
```

Figure 14 Directory structure for user recommendation build process
The parameters needed by the script are highlighted by red, blue and yellow color in the above command.
The first parameter (in green) is the path of the input file on HDFS.
The second parameter (in red) is the path of the folder on HDFS where the output files will be written by the spark job.
The third parameter (in yellow) is the threshold used by the similarity computation implementation of the spark library.

6.2.4 Processing the output file
To upload the results on the HBase column family, we need to process the results to comply with the input file format needed by the HBase TSV file import module. Same as document recommendation, we use a python script (ParseResults.py) to make it comply with the TSV format needed by HBase file importer. Finally, our data is in a format as shown in the Figure 16.

Figure 15 Screenshot of final output for user recommendation

6.2.5 Upload to HBase
Once we process all the output files, we can upload them to HBase using the following command:

```
usage: hbase org.apache.hadoop.hbase.mapreduce.ImportTsv -Dimporttsv.columns=HBASE_ROW_KEY, user Cf:documentID, user Cf:simScore

ideal/cs5604s16-user/user/cs5604s16_cf/parseUser.txt

```

The red colored text are the two column families where our data is being uploaded. The blue colored text is the name of the table name in HBase. The yellow colored text is the processed output file.

```
ROWKEY sim_tweet_1, sim_tweet_2, sim_tweet_3 score_1, score_2, score_3
```

The format of the line in the processed output file is as below:
Note that the fields in the line are tab separated.
7 Lessons Learned

Our implementation described in report 2 consisted of 2 MapReduce jobs that are described in the next subsections. We have replaced this design with a better design based on Apache Spark’s machine learning library (MLlib) packages as described in the previous sections.

7.1 SIMILARITY COMPUTATION ALGORITHM

Currently, in similarity calculation, we separate tweets from webpages. For tweets, we only calculate similarity between different pairs of tweets. If a search result is a tweet, we will recommend similar tweets based on its contents. If a search result is a webpage, we recommend similar webpages based on the contents of webpages. We are not comparing the similarity between tweets and webpages for now.

We are using term frequency–inverse document frequency (TF-IDF) weights to evaluate cosine similarity between document pairs.

We first remove the stop words in each tweet and get arrays of terms for each tweet. Then we calculate the term frequency for each term in each tweet and normalize the result by dividing the number by the length of the cleaned tweet. For example, the term frequency of term1 in tweet 1 is 1, and the normalized result is 0.2. (Figure 16)

![Tweet input file](image)

*Figure 16 Tweet input file*

Given the arrays, we are able to build a list of terms that appear in the whole tweet collection. For each term, we record the tweets in which it appears, thus we have the document frequency as well. (Figure 17)

In order to save the computation time, for each tweet, we find a group of tweets that has at least one term in common with it, and only find the k most similar tweets from those tweets. (Figure 18)
Now, for each tweet we are recommending upon similarity score, we calculate the \(tf-idf\) score for each term against the tweets in the group we mentioned above. Each tweet in the group can now be viewed as a vector of \(tf-idf\) scores for the terms in this current tweet, so we can compute cosine similarity between them, sort the cosine product and find the \(k\) tweets with largest value.

<table>
<thead>
<tr>
<th>Term 1</th>
<th>Term 2</th>
<th>Term 4</th>
<th>Term 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.281234684152</td>
<td>0.281234684</td>
<td>0.374979579</td>
<td>0</td>
</tr>
<tr>
<td>0.325257498916</td>
<td>0.325257499</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0.325257498916</td>
<td>0</td>
<td>0</td>
<td>0.433676665</td>
</tr>
</tbody>
</table>
In order to demonstrate our result, we run the algorithms on the first 9 tweets in the cluster:

![First nine tweets in the cluster, for testing purpose](image)

And here is a matrix of the tweet similarity:

![Tweets similarity matrix (of first nine tweets)](image)

### Figure 22 Tweets similarity matrix (of first nine tweets)

#### 7.2 MAPREDUCE

Document similarity calculations theoretically take \( O(n^2) \) time if executed serially. However, we have implemented 2 MapReduce jobs described next. These MapReduce jobs will execute the code on different mappers and reducers in parallel and hence would reduce the time complexity to \( O(n \times m) \), where \( n \) is the total number of documents and \( m \) is the potential similar documents for each document which will be a small fraction of \( n \).

In the current state of the project, the pipeline of execution looks like Figure 23
The output file will contain, for each tweet, potentially similar tweets that can be recommended to the user on a ‘more like this’ page if that tweet appears in the search results of the query. For example, for tweet id 1 (tid 1 in Figure 23), tweet 2, 3 and 4 (tid 2, tid 3 and tid 4) can be recommended as similar tweets to the user, based on the document similarity. However these tweets need to be ranked based on their exact cosine similarity before displaying the recommendations.

7.2.1 MR-1:
The input to the first mapper will be the file containing tweet IDs and content of those tweets, with one tweet per line. The tweet ID and its content will be separated by a space. The mapper process will read one line (tweet) at a time from standard input. We are using the Hadoop streaming API, which reads from the tweet input file and feeds them to the standard input of the different parallel map tasks. It will tokenize the line into two parts, the first being tweet ID and second part being the tweet content. It will further tokenize the tweet content to extract all the terms. It will then print multiple key-value pairs for each tweet to standard output. Each key-value pair will be as shown in Figure 24. The keys will be the lemmatized terms from the tweet content and if this term is not a stop word only then will the mapper print its key-value pair, value being the tweet ID.
The Hadoop MapReduce framework will sort the emitted key-value pairs for all the tweets. Hence all the keys with the same value will be grouped together internally in the framework before the reducer process is called. The framework will feed these sorted key-value pairs to the standard input of the reducer processes. Hence the reducer process of the first MapReduce job will see key-value pairs having the same keys, in sequential fashion. The reducer process will append all the values (tweet IDs) for a single key (term) in a comma separated values (CSV) format and emit a key-value pair where the key will be the term and the value will be the tweet IDs in CSV format as shown in Figure 24.

7.2.2 MR-2:
We use the output of the first MapReduce job as the input to the next MapReduce job. We will again use Hadoop streaming API to read the key-value pairs that were produced by the reducer of the first MapReduce job. Hadoop streaming will feed one line at time from the output of the first job to the standard input of the mapper process of the second job. This second mapper process will tokenize all the tweet IDs from the value fields and emit all the combinations of the tweet IDs along with mirror pairs as shown in Figure 25.
The Hadoop framework will sort these key-value pairs according to their key values and hence all the key-value pairs having same tweet ID will be grouped together. The framework will then feed these sorted key-value pairs to the standard input of the reducer processes. The reducer process will group together all the value fields (tweet IDs) with the same key values (also tweet IDs) in CSV format and print to standard output the key-value pairs where key will be the tweet ID and value will be a CSV formatted string having other tweet IDs having certain terms in common with the tweet ID of the key field. This final output of the reducer of the second MapReduce job is shown in Figure 26.

**Figure 25 Output of mapper of MR-2**

**Figure 26 Output of reducer of MR-2**

### 7.3 MAPREDUCE CODE

Even though we are not using this approach, we have uploaded our working code to the following location:

[https://drive.google.com/open?id=0B1ivYOQz7rF1YndBZXF6TV8xcG8](https://drive.google.com/open?id=0B1ivYOQz7rF1YndBZXF6TV8xcG8)

Files included -

- isrmapper1.py – Mapper for first stage, outputs term-document (k-v) pairs
- isrreducer1.py – Reducer for first stage, aggregates all terms and documents which contains them as the csv separated value (term - doc1, doc2, ... doc_n)
• isrmapper2.py – Mapper for second stage, outputs document-document (k-v) pairs
• isrreducer2.py – Reducer for second stage, aggregates document IDs and other documents with similar terms (doc1, doc2, doc3, …)

8 Challenges

The first challenge is that there will be no user information at all in our information retrieval system at the beginning; chances are that most of the users will be guests and won’t have long-term login history for us to keep track of. Hence, we won’t be able to call Twitter’s API and request a certain user’s history from Twitter and make personalized recommendations. This forces us to focus on ad-hoc recommendations, start with item-based algorithms and hybrid with collaborative filtering as we get more user data.

We have talked to the Solr team and Front End team and requested from them user information like session, query history, search results, viewing history, clicks and so on. We might not be able to get all that information, but anything we get from those teams, will be added to the user vector and used to calculate user similarity for recommendations. Hence, we design our work as follows:

1. The first user visits our system and makes the first query
2. We recommend to this user documents that are similar to the search results (using item-based clustering)
3. Starting from the second user, we calculate the similarity between the current user and previous users. If they are similar enough (above a certain threshold) we will hybrid with user-based algorithms to make better recommendations.

There are a lot of recommendation systems in the e-commerce industry, from which we can learn the algorithms and adapt some of their features to our concerns.

After the first two development phases, we realize that it requires more work than expected on Front End team’s side to provide user information to us. Such being the case, we will take our GRAs’ suggestions to generate some random user data as a starting point to build user-based recommendation system.

Another challenge is we are getting dynamic results from the Solr team, and not having used Solr before. We will also need to talk to the Solr team and Front End team to discuss more about our input and output format, so that we can embed our software into the overall system.

To tackle this challenge, we have migrated to Apache Spark to do all the implementation, and as mentioned above, will bypass the format issue by generating temporary random user data. In the next phase, we are looking to explore the MoreLikeThis handler provided by Lucene.
Last but not least, we also need to decide how many recommendations to make to users. An intuitive solution is to adjust this number according to user’s choice, but we will need to read previous user studies, as well as consider computation time to make a reasonable estimation.

For this challenge, we are storing all the similarity computation results in comma separated value (csv) format and store them in two columns in HBase, so that the other teams can decide how many recommendations to use by themselves.

9 Evaluation

9.1 USER-BASED RECOMMENDATION
Although we are not working on real user data, we did an initial evaluation on our recommendation system on the training set to have a general idea of the system performance. We computed the rating score of each (user, document) from our self-generated fake user data (training set) and computed the mean squared error. The error value has order of magnitude $10^{-4}$.

Because the user data is faked, the user’s clicking history are made of random tweets. It is hard for us to find ways to evaluate the result. Once we have the real user in the system, we can do further evaluation.

9.2 ITEM-BASED RECOMMENDATIONS
We used mean squared error for these recommendations as well. As stated in the user manual section, we are using a minimum threshold as an input to the DIMSUM algorithm. This threshold produces estimates of the column similarities but its correctness is guaranteed for those column pairs whose threshold is greater than the minimum threshold. To do this, we compute the absolute column similarities as well and compare the results with the results obtained by providing minimum threshold.

10 Future Work
In this project, we completed 2 types of recommendations:
1. Item-based recommendations
2. User-based recommendations
There can be a third type, which is the hybrid of the two approaches. This approach might produce more accurate recommendations to enhance the user interaction with the system. This approach will combine the user preferences as well as the items frequently purchased together. The recommendation system built using this hybrid approach is more likely to be used in the industry because of the increased user satisfaction. We believe this is a great direction for the future
of this project, since we have achieved the first 2 basic types of recommendations. However the only road block can be the lack of genuine user browsing history, since in the class setting such as this class, very few people would actually browse documents in the system to generate browsing history.

11 Acknowledgements


Besides that, we want to thank Dr. Fox and our GRAs Mohamed Magdy Farag and Sunshin Lee for guiding and advising us. We want to thank NSF for grant IIS - 1319578, Integrated Digital Event Archiving and Library (IDEAL). We are also grateful for the effort by the Front-end and Solr team to communicate and provide the data we need to implement the system. Last but not least, we thank the whole class for discussing the problems and learning about information retrieval together.

We will keep updating this section when we finished and pay respect to all the people and resources that helped us through the semester.

12 Timeline

<table>
<thead>
<tr>
<th>Time</th>
<th>Work Planned</th>
<th>Work Progress</th>
</tr>
</thead>
<tbody>
<tr>
<td>Week 1</td>
<td>Reading Chapter 11 of a new book by Zhai and Massung. Getting background knowledge about collaborative filtering</td>
<td>Done</td>
</tr>
<tr>
<td>Week 2</td>
<td>Reading other papers related to the project</td>
<td>Done</td>
</tr>
<tr>
<td>Week 3</td>
<td>Write literature reviews -- each of us write a version and combine it</td>
<td>Done</td>
</tr>
<tr>
<td>Week 4</td>
<td>Follow the tutorials posted on Canvas and get familiar with the techniques that will be used. (Emphasize on learning Spark) Come up with preliminary plan for the whole project</td>
<td>Done</td>
</tr>
<tr>
<td>Week 5</td>
<td>Learn Solr and Spark in more detail and talk to Solr team &amp; classification team. Modify our plans and start initial development</td>
<td>Done</td>
</tr>
<tr>
<td>Week 6</td>
<td>Use script to download webpages mentioned in tweets. Environment setup (Hadoop, Mahout). Test Mahout recommendation tool.</td>
<td>Done</td>
</tr>
<tr>
<td>Week 7</td>
<td><em>Spring Break</em></td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>----------------</td>
<td></td>
</tr>
<tr>
<td>Week 8</td>
<td>Finish doing content-based recommendation for example data and evaluation</td>
<td>Done</td>
</tr>
<tr>
<td>Week 9</td>
<td>Generate random user information; start building user-based recommendation</td>
<td>Done</td>
</tr>
<tr>
<td>Week 10</td>
<td>Finish doing user-based recommendation for faked user data</td>
<td>Done</td>
</tr>
<tr>
<td>Week 11</td>
<td>Modification for getting better result; tailor implementation for updated user data format</td>
<td>Done</td>
</tr>
<tr>
<td>Week 12</td>
<td>Accuracy evaluation and modification</td>
<td>Done</td>
</tr>
<tr>
<td>Week 13</td>
<td>Presenting the final project outcome</td>
<td>Done</td>
</tr>
<tr>
<td>Week 14</td>
<td>Review the report</td>
<td>Done</td>
</tr>
<tr>
<td>Week 15</td>
<td>Final review, modification and evaluation</td>
<td>Done</td>
</tr>
<tr>
<td>Week 16</td>
<td>Submitting the final report and associated files</td>
<td>Done</td>
</tr>
</tbody>
</table>

*Table 2 Tentative timeline of the project*

13 References


[3] Zhai and Massung, Chapter 11 Recommendation systems, pages 197-211


