A COMPARISON OF COLLABORATIVE FILTERING ALGORITHMS FOR JOB RECOMMENDATION USING APACHE MAHOUT

A Project

Presented to the

Faculty of

California State Polytechnic University, Pomona

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

In

Computer Science

By

Chantal Fry

2016

SIGNATURE PAGE

PROJECT:	A COMPARISON OF COLLABORATIVE FILTERING ALGORITHMS FOR JOB RECOMMENDATION USING APACHE MAHOUT
AUTHOR:	Chantal Fry
DATE SUBMITTED:	Fall 2016
	Computer Science Department

Dr. Yu Sun
Project Committee Chair
Computer Science

Dr. Daisy Sang
Computer Science

ACKNOWLEDGEMENTS

I would like to thank my parents, Jules and Diane Pellaers, who have helped me through my tough times throughout the years and always encouraged me to keep going with my education. If it weren't for them I may not have made it here.

I also extend sincere thanks to my advisor and co-advisor, Dr. Yu Sun and Dr. Daisy Sang, who both helped encourage me to keep going on this project and provided me with much needed help, support and guidance. Additionally, thank you to the graduate support staff including Dr . Young and Annie Montes who also helped provide help along the way.

Finally, I would like to thank the many co-workers at both the Jet Propulsion Laboratory in Pasadena, CA, and Lawrence Livermore National Laboratory in Livermore, CA who helped debug my code and gave me hints on where to go next with this project throughout the many months I have worked on it. I owe you endless thanks and coffee!

ABSTRACT

Recommender systems were developed to assist users in making choices when presented with a large number of selections. Well known for their use in e-commerce, they have now become common throughout many other application areas including content providers, e-learning, and social media websites.

This project explores the problem of predicting job postings to users on a professional social networking website, XING. The Java Apache Mahout framework along with an Apache Solr data storage solution is used to implement a content-based recommendation algorithm and various iterations of collaborative filtering recommendation algorithms. The algorithms are then evaluated using an evaluation framework combining precision, recall, and user success metrics to compare their performance. Results indicate that collaborative filtering algorithms perform better than a pure content based recommender. However, although results in this project did not indicate that a multimodal recommender performed better than a collaborative filtering recommender, literature indicates that, with work, this type of system has the potential to provide superior recommendation results.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Recommender systems are programs which use information about users such as user interests and locations in order to attempt to recommend, through the use of some type (s) of recommendation algorithms, items that the user is likely to find relevant or useful [16] [44] [45]. Users of recommender systems can be individuals or entire businesses. Items recommended by these systems can include both products (movies; music; products) or services (financial; insurance; transportation).

With the popularity of the Internet, computers, and smartphones, recommender systems have become so ubiquitous that we often no longer realize their presence. Applications and websites in areas ranging from e-commerce (Amazon; eBay; Alibaba) to e-learning (blended learning [32]; distance learning programs [54]) to social media [34], [46] (Facebook Top Stories feed; Twitter Best Tweets feed) leverage recommender systems for displaying information and products to users [44].

The purpose of recommender systems is to aid users in selections. Without them, the amount of data available is huge, overwhelming, and impossible for even a power user to peruse unassisted. Conventional search can help a user find products or services; however, this does not help the user find new products and services that they may not have previously known about. Leveraging recommender systems to make this data more organized and visible is beneficial to both the user and the provider of the service. For example, in e-commerce, if a user is recommended a new product that they think is interesting and in turn purchase it, the seller makes a profit.

An interesting problem in the field today is in the area of job posting recommendation. With an increasing number of companies turning their hiring efforts to websites such as LinkedIn, Monster and XING, there can be an extremely large amount of job postings for an applicant to sift through. For example, entering a search on LinkedIn for ”Software Engineer” in ”San Francisco, CA” returns 124,913 results as of May 16, 2016. This amount of postings is much too overwhelming and time consuming for an individual to go through.

A recommendation system with a well-implemented algorithm can help alleviate this problem by picking out job postings that are most relevant to the given user and displaying them to the user in a special interface as shown in Figure 1.1. This both alleviates a user’s need to manually perform a search query and a their feelings of being overwhelmed with the enormous amount of results that can match a simple query for job postings.

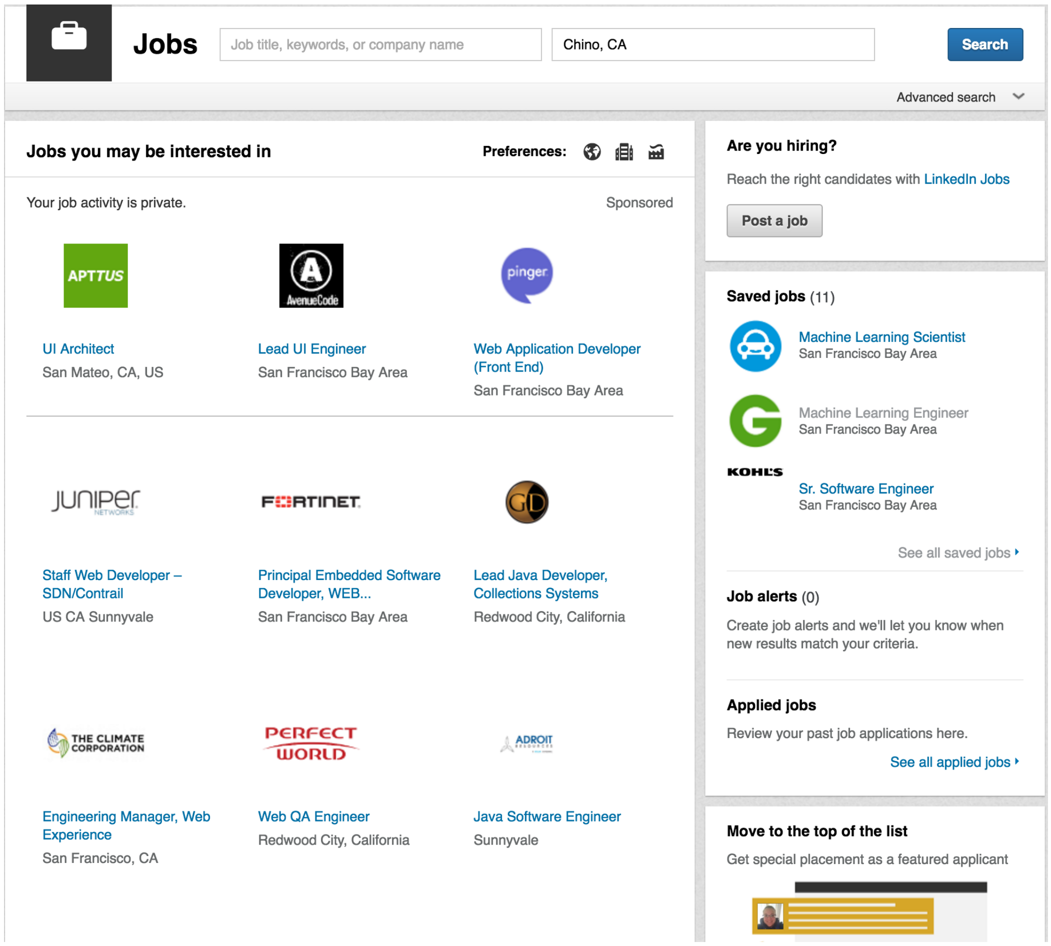
1.1 What are Recommender Systems

Formally, recommender systems, also commonly called recommendation systems, are software systems designed to solve the problem of estimating user ratings, or preferences, for items that the user has not yet seen.

Given a set of users U, a user u such that $u \in U$, a ratings scale R, and a set of items I, a recommender system aims to solve the following problem: Based on the knowledge of prior preferences of user u and users similar to u, predict the items, i_1, i_2,...i_j, where $i_1, i_2, \ldots i_j \in I$, that u would indicate highest preference for based on ratings scale R.

This can be also viewed as a formal optimization problem where the quantity to be optimized is the rating. This can be expressed as follows in Equation 1.1:

$$\forall u \in U, i_u = \arg\max_{i \in I} r(u,i) \tag{1.1}$$



Jobs you may be interested in

Your job activity is private. Sponsored

UI Architect San Mateo, CA, US	Lead UI Engineer San Francisco Bay Area	Web Application Developer (Front End) San Francisco Bay Area
Staff Web Developer – SDN/Contrail US CA Sunnyvale	Principal Embedded Software Developer, WEB... San Francisco Bay Area	Lead Java Developer, Collections Systems Redwood City, California
Engineering Manager, Web Experience San Francisco, CA	Web QA Engineer Redwood City, California	Java Software Engineer Sunnyvale

Are you hiring?

Reach the right candidates with LinkedIn Jobs

Saved jobs (11)

Machine Learning Scientist
San Francisco Bay Area

Machine Learning Engineer
San Francisco Bay Area

Sr. Software Engineer
San Francisco Bay Area

See all saved jobs

Job alerts (0)

Create job alerts and we'll let you know when new results match your criteria.

Applied jobs

Review your past job applications here.

See all applied jobs

Move to the top of the list

Get special placement as a featured applicant

Figure 1.1: Example of displaying job posting recommendations (LinkedIn)

A high-level model of user interactions with a typical a recommender system is depicted in Figure 1.2. Consisting of five major steps, summarized as follows:

1. User interacts with the interface of a recommender system. User preferences are obtained by an explicit profile and user actions.

2. User preference information is fed into the recommender system.

3. A series of recommendation techniques and algorithms uses this preference information to calculate a series of personalized recommendations for the user.

4. Any additional logic specific for the given application is applied.

5. The personalized recommendations are displayed to the user.

6. User interacts with the updated user interface.

This simplistic, high-level model fails to address many of the issues that commonly plague recommender systems such as the cold-start problem, the new user problem, and the new item problem.

These issues will be explored in Chapter 2.

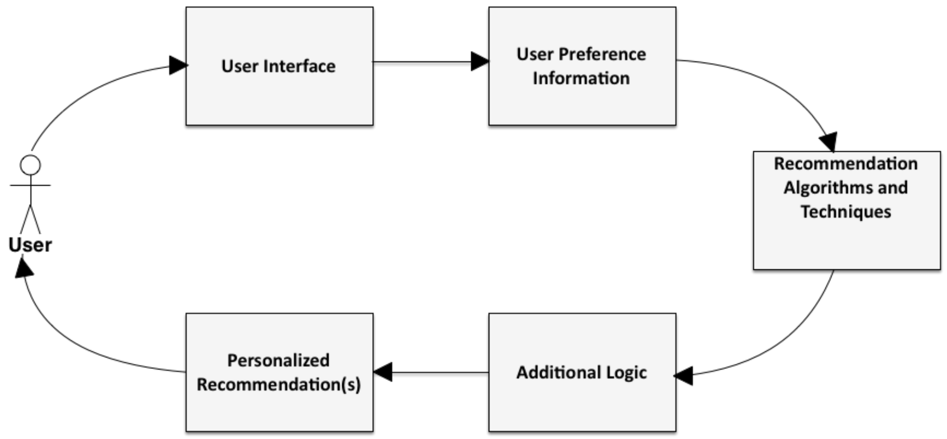

User Interface
User Preference Information
Recommendation Algorithms and Techniques
User
Personalized Recommendation(s)
Additional Logic

Figure 1.2: Diagram of high-level control flow of typical recommender system.

1.2 Project Summary

This project will compare the performance of a variety of collaborative filtering algorithms in solving a job recommendation task. Using a pseudo-anonymized dataset from the professional social networking site XING, for each of 150,000 users, we will predict up to 30 of the most likely job postings that each user is likely to look at, in ranked order of likeliness for each of the following algorithms:

- Content-based (no collaborative filtering)
- Item-based collaborative filtering with Tanimoto Coefficient Similarity
- Item-based collaborative filtering with Log Likelihood Similarity
- User-based collaborative filtering with Tanimoto Coefficient Similarity for each of 2, 16, 128, 1024 Nearest Neighbors
- User-based collaborative filtering with Log Likelihood Similarity for each of 2, 16, 128, 1024 Nearest Neighbors
- Multimodal (Hybrid) Recommender with Log Likelihood Similarity for 512 Nearest Neighbors plus Content Indicators

The performance of each algorithm will be based on a pre-existing evaluation framework provided online. This framework scores each algorithm based on a mix of several accepted common evaluation measures for recommendation systems such as precision, recall, and user success; it will be described in detail in Section 3.3. The best performing algorithm will be determined by the algorithm that receives the highest score from the evaluation algorithm.

1.3 Paper Organization

The remainder of this paper will be organized as follows. Chapter 2 will begin by providing a comprehensive overview of background information on recommendation systems through a literature review. Next, Chapter 3 will formally define the problem approached in this project. This will include a formal explanation of the problem as well as information about the datasets and algorithms commonly used to solve the problem. Chapter 4 will then summarize the methodology and approach we took to solve this problem for the purpose of this project. Chapter 5 will then provide a review and explanation of the results achieved through the methods described in Chapter 4. Finally, Chapter 6 will serve as the concluding chapter for this report, once again briefly summarizing the results obtained as well as addressing potential avenues for further work and improvement.

Chapter 2

Literature Review

This chapter serves to provide background information on recommender systems. Although not comprehensive, it serves to provide an adequate foundation for understanding the remainder of this work.

2.1 Recommender System Fundamentals

This section presents general information common to recommender systems. The user-recommender lifecycle will be discussed, followed by a presentation on user interfaces and its importance for the success of a recommender system. Then components, considerations, and common challenges that must be noted when designing a recommender system will be listed and described.

2.1.1 User Interfaces

Psychological studies have shown that users actually have a difficult time mapping their complex system of thoughts, experiences, and insights into scale based rating systems. This process is complicated further when users do not rate products in a timely manner [26]. This can result in poor data quality that can adversely affect the performance of a recommender system [55]. Taking care in researching and properly designing an effective user interface can help improve this problem [48]. This user study found that providing cues to users such as exemplars can help users rate products more consistently, improve usability, and provide higher data quality.

A complete review of user interfaces and how they relate to recommender systems is beyond the scale of this project.

2.1.2 Components and Considerations

There are several components and considerations required for building a recommendation system that are independent of the application [32] [45]. These include:

- Dataset
- Filtering technique
- Data model
- Sparsity of dataset
- Scalability of data storage to the dataset
- Time and memory consumption of system
- Objective of results
- Quality of results

The decisions in these areas will greatly affect the behavior and performance of the recommendation system. The type and size of the dataset, for instance, should be considered to ensure the ideal storage system is used that will provide adequate scalability for the desired application. For research and testing of recommender systems in development, there are many sample datasets available that could be useful in a variety of applications.

A compilation of some of these datasets and their useful properties can be found in Table 2.1 [27]. The choice of filtering technique refers to the type of algorithm selected in order to recommend results to the user [45]. The choice of filter technique depends on the objective (recommending top-N movies; recommending add-on purchases) and the quality goals (accuracy; novelty; coverage; precision) for the algorithm results. This will be further discussed in Section 2.2.

Table 2.1: Sample of Publicly Available Datasets

Dataset	**Users**	**Items**	**Ratings**	**Density**	**Rating Scale**
Movielens 1M	6,040	3,883	1,000,209	4.26%	[1,5]
Movielens 10M	69,878	10,681	10,000,054	1.33%	[0.5, 5]
Movielens 20M	138,493	27,278	20,000,263	0.52%	[0.5, 5]
Jester	124,113	150	5,865,235	31.50%	[-10, 10]
Book-Crossing	92,107	271,379	1,031,175	0.0041%	[1, 10], implicit
Last.fm	1,892	17,632	92,834	0.28%	Play counts
Wikipedia	5,583,724	4,936,761	417,996,366	0.0015%	Interactions
OpenStreetMap	231	108,330	205,774	0.82%	Interactions
Git (Django)	790	1,757	13,165	0.95%	Interactions

2.1.3 Challenges when Designing Recommender Systems

Challenges when developing recommender systems span a wide range of issues. Major areas, including a short description of the challenges contained in each are:

1. *Sparse data* [35] [44] [45]: In a recommender system with a large and diverse amount

of data, there is little overlap between users and uneven distribution of user ratings on data items (e .g . movies, books, news feed stories etc).

2. *System scalability* [35] [44] [45]: Recommender systems can start out with an average size data-set and a small user base. As popularity increases, the user base is likely to increase and the dataset size may also increase. Thought must be given into the selection of hardware and recommender algorithms to be in line with the growth of the system.

3. *Cold start problem* [15] [35] [44] [45]: The cold start problem occurs when a new system is deployed, a new user begins using a system, or even with established users who only use a system rarely. Without prior data, or with changed user interests, the recommender system may fail to recommend any useful information for the user.

4. *Diversity vs. Accuracy problem* [44] [45]: The user of a recommender system should receive a diverse range of results while still providing results that align with the preferences of the specific user.

5. *Security* [19] [45]: Recommender systems are vulnerable to a variety of malicious attacks including characterizing attacks, low-knowledge attacks, nuke attacks, and informed attacks. Measures to protect systems against these attacks must be considered.

6. *Privacy* [35] [36]: Recommender systems may use personal information of users when generating recommendations such as geographical location, movie genre preferences, and previous purchases. Users expect this information to remain within the scope of the recommender system.

7. *Evaluation metrics* [45]: Developers of recommender systems must take care to choose the best metrics to evaluate a given recommender system. The metrics used for one recommender system are not necessarily appropriate for another.

8. *User experience* [23] [38] [45]: Many aspects of a recommender system can have an effect on user experience. These include not just the user interface, but also the accuracy, diversity, ease of product selection, perceived system effectiveness, and user satisfaction with results of choosing a recommended product.

9. *Overspecialization* [35]: This is a problem common to content-based recommender systems where the recommendations to the user do not contain an adequate degree of novelty or serendipity.

2.1.4 Similarity Measures

Similarity measures are used in recommender systems to determine the similarity between items and/or users within a system. This is used most often for collaborative filtering and hybrid systems which incorporate aspects of collaborative filtering [16]. Similarity measures are also commonly used for certain evaluation metrics of recommender systems [60] [68] [71]. Although the similarity metrics here have been presented in the context of users, they can easily be adapted to be used in the context of items. Commonly used similarity measures and their definitions are [5] [21] [17] [56] [22] [45]:

- *Pearson Correlation*:

$$PCC_{i,j} = \frac{\sum_{u \varepsilon U} (r_{i,u} - \bar{r_i})(r_{j,u} - \bar{r_j})}{\sqrt{\sum_{u \varepsilon U} (r_{i,u} - \bar{r_i})^2 \sum_{u \varepsilon U} (r_{j,u} - \bar{r_j})^2}} \tag{2.1}$$

  where U is the set of all items that both user i and j have rated, r is the actual rating and r is the predicted rating.

- *Cosine Similarity*:

$$\cos(\vec{i}, \vec{j}) = \frac{\sum_{u \varepsilon U} r_{i,u} r_{j,u}}{\sqrt{\sum_{u \varepsilon U} r_{i,u}^2 \sum_{u \varepsilon U} r_{j,u}^2}} \tag{2.2}$$

where U is the set of all items that both user i and j have rated and r is the actual rating.

- *Adjusted Cosine Similarity*:

$$cos(\vec{i},\vec{j}) = \frac{\sum_{u \varepsilon U}(r_{i,u}-\bar{r_u})(r_{j,u}-\bar{r_u})}{\overline{\sum_{u \varepsilon U}(r_{i,u}-\bar{r_u})^2} \ \overline{\sum_{u \varepsilon U}(r_{j,u}-\bar{r_u})^2}} \tag{2.3}$$

where U is the set of all items that both user i and j have rated, r is the actual rating and r_u is the average of the u^{th} users ratings.

- *Mean Squared Difference*:

$$MSD = \frac{1}{|U|}\sum_{u \varepsilon U}(r_{i,j}-r_{j,u})^2 \tag{2.4}$$

where U is the set of all items that both user i and j have rated and $|U|$ is the total number of users.

- *Euclidean Distance*:

$$euc_{i,j} = \sqrt{\sum_{u \varepsilon U}(r_{i,u}-r_{j,u})^2} \tag{2.5}$$

where U is the set of all items that both user i and j have rated.

- *Log-Likelihood Ratio*:

$$\lambda = \frac{\max_{w \varepsilon \Omega_0} H(\omega;k)}{\max_{w \varepsilon \Omega} H(\omega;k)} \tag{2.6}$$

where Ω is the parameter space and Ω_0 is the hypothesis being tested. The entire ratio function represents the ratio between the maximum value of the likelihood function in the neumerator to the maximum value of the likelihood function over the entire subspace. The final ratio value is a prediction of how likely it is that the values, or in this case, users are actually related to each other.

- *Tanimoto Coefficient (also known as Jaccard Coefficient)*:

$$T(r_{i,u}, r_{j,u}) = \frac{r_{i,u} \cap r_{i,j}}{(r_{i,u} \cup r_{i,j})} \tag{2.7}$$

  where $r_{i,u}$ and $r_{j,u}$ are the set of recommendations for user i and j, respectively. The Tanimoto Coefficient attempts to measure the degree of overlap between two datasets.

2.2 Recommender Systems Algorithms

Although there are a vast number of algorithms used in recommender systems, the literature focuses on three main techniques: (1) Content based recommendation techniques; (2) Collaborative filtering recommendation techniques; and (3) Hybrid techniques that are most commonly some combination of both content based and collaborative filtering. Hybrid techniques may additionally make use of combinations of any of the other techniques mentioned in this section. Content based, collaborative filtering, and hybrid algorithms are discussed in detail in their corresponding sections.

Additional techniques mentioned in current literature such as social based filtering, context aware filtering, knowledge-based filtering and computational intelligence filtering are discussed in their corresponding sections below in less detail.

2.2.1 Content based recommendation techniques

Content based recommender systems make recommendations based on past user selections [16] [44] [45] [52] [64]. In these systems, each user has their own personalized model from which recommendations are generated. Content can be generated from analyzing data objects, therefore features of audio files, video files, and text can be used as content from which to use to generate recommendations. Often, content based recommenders return results in a ranked, top-n format based on a similarity score, similar to the classic information retrieval search problem [25].

Although in general, content based recommender systems are known for their ease of implementation, for some types of content feature extraction may be difficult [14].

2.2.2 Collaborative filtering based recommendation techniques

Collaborative filtering based recommender systems uses user ratings and preferences information to make recommendations [16] [44] [45] [52] [57] [62]. These user ratings can be acquired either explicitly (i.e. a user rating a movie five stars) or implicitly (i.e.ȧ user listening to a song 100 times).

There are two different ways to categorize collaborative filtering systems. The first of which is whether they are memory-based or model based [12]. Memory-based systems use information from users similar to the target user. They will use preference information from the similar users to create recommendations for the target user. This type of model is vulnerable to a variety of profile injection attacks, where fake users in the database skew the quality of the recommendations generated by the system [19]. Conversely, model-based systems identify patterns in user preferences for the target user and use this information to generate new recommendations.

Memory based models commonly use algorithms such as k Nearest Neighbors (kNN) and other classical top-N-based algorithms for performing collaborative filtering. These algorithms are typically easy to implement. In either of these methods, similarity computations are used such as Cosine Similarity or Pearson Correlation Similarity to compute similar users.

There are several issues common to this collaborative filtering model: (1) concentration bias, the phenomenon that the same popular items are the ones that typically come up as recommendations; (2) overspecialization, the issue of recommender systems recommending items very similar to what the user has already consumed. It has been shown by [4] that these problems can be improved upon by using improved probabilistic k-nearest neighbor (kPN) algorithms as opposed to the more classical base algorithm (kNN).

Model based approaches are required to learn from user preferences. This can be done using techniques from machine learning or data mining. For these approaches, dimensionality reduction techniques are very popular approaches [45]. One of the most popular techniques used for this today is matrix factorization [57] [58] [70]. This is commonly implemented through techniques such as Stochastic Gradient Descent or Alternating Least Squares [28] [40] [57] [69].

Single-machine implementations of these model based algorithms can suffer issues with efficiency and scalability. Due to the increasingly large size of the datasets, and thus increasing dimensionality of the corresponding matrices used to represent user-item preferences, scalable and distributed methods of matrix factorization are increasingly necessary. Scalable and distributed matrix factorization can be achieved using techniques such as MapReduce and Hadoop [49] [57].

A second method of categorizing collaborative filtering recommender systems is based on whether their algorithm produces recommendations through user-based techniques or item-based techniques. In user-based collaborative filtering, a prediction is made for a user's interest in an item based on rating information or interaction information from similar users [49] [63]. In an item-based collaborative filtering system, these recommendations are instead obtained by finding similar items to those the test user has already rated [49] [63]. Item based recommendations can be faster in implementation because since item ratings are generally more static, recommendations can be precomputed periodically [49]. However, if item ratings are very sparse, the results may not be ideal.

Collaborative filtering based recommender systems additionally are vulnerable to the cold start problem [15] [16] [44] [45]. There are three different versions of the cold start problem: (1) new user; (2) new item; and (3) new community [16] [51]. The commonality between all three versions is that when something new is introduced into the system, little is known about it or its relationship to other objects in the system. Therefore, it is difficult for the system to initially reconcile how to use the new user (s) or item (s) when creating recommendations. This may result in poor recommendations, specifically in the case of a new community or new user. In a pure collaborative-filtering based system, quality of recommendations can be improved by incorporating information based on other users (given that there are enough in the system) [16]. Using more sophisticated techniques, such as hybrid recommendation systems, which will be discussed in Section 2.2.8, opens opportunities for novel ways to resolve the cold start problem.

2.2.3 Context-aware recommendation techniques

Many recommender systems may perform well under typical circumstances, but fail to adapt their recommendations to changing contexts [11] [12] [30] [33]. This can cause poor recommendations for users under certain situations. Due to these facts, context-aware recommender systems are of high interest in todays literature [35].

In the literature, context is typically defined using Anind and Abowds definition [3]: Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves.

Additionally, context can be divided into three types. These type categories are [11] [12]:

- *Computing Context*: network connectivity, communication costs, communication bandwidth, workstations, printers, scanners
- *User Context*: location, user, social affinity, social situation, people nearby
- *Physical Context*: lighting, traffic conditions, temperature

For a concrete example, consider a music recommender system application and a user [33]. The users music preference may change depending on the context. If it is at night, the user may prefer to listen to soothing music while during the day they may prefer to listen to rock music. While working out, the user may prefer to listen to upbeat pop music, while during working hours they may prefer to listen to classical music. If the recommender considers these factors (time of day and/or user activity) while populating recommendations, it is making use of context. Thus, it is context-aware.

Algorithms used in context-aware recommender systems include Contextual Pre-filtering, Contextual Post-filtering, Contextual Modelling, Hidden Markov Models, Sequential Pattern Mining, Item-based Markov Modeling, User-based kNN, and Matrix Factorization with Bayesian Personalized Ranking [11] [30] [33].

2.2.4 Knowledge-based recommendation techniques

Knowledge based recommenders use knowledge about users and and objects in the system to aid in determining recommendations that may better meet user requirements. This approach can perform better in more complex domains [43] [53]. Ontologies are often used to represent the relationships between users and recommendations [16] [47].

These types of recommender systems have some advantages and disadvantages to their use [43]. Advantages of knowledge based systems include the elimination of the cold start problem as ratings information is not required in the recommendation process and elimination of the need to store large amounts of user information. However, some disadvantage include more static ratings lists, and that they are difficult to implement due to the need for a complex understanding of knowledge engineering.

2.2.5 Social based recommendation techniques

Social networks are ubiquitous in our daily lives. Popular social-networking communities such as Facebook, Twitter and LinkedIn have billions of active users. We use them for many daily activities, tracking our past and future activities, thoughts and feelings, and keeping in touch with friends, family and acquaintances. Together, this large web of interconnected relationships created in a social network as shown in Figure 3, provides a large amount of information that could be exploited.

This amount and type of information (relationships, interests, location etc) makes the integration of social based techniques into recommender systems an intriguing concept. Additionally, it has been shown that users tend to trust recommendations more when they come from family or friends, as this increases their perceived trust in the suggestions [16] [34] [66]. Thus, exploiting this information can result in, from a users perspective, a better-quality recommendation system.

Issues that may come up with a social based recommendation system data sparseness and scalability issues [34]. Scalability issues can arise due to algorithm choices. For example, if matrix factorization techniques are used, every time a new user joins the social network, a new column will be added to the matrix, as interests change the matrix will change and each time the matrix will need to be refactored. With a social network with billions of users, this can be a time-consuming process with poorly designed algorithms and architecture.

2.2.6 Demographic based recommendation techniques

Demographic based techniques incorporate demographic information such as age, gender, income, and degree level in order to provide recommendations [16] [50] [51]. This approach works because groups of people with similar demographic information tend to have similar interests [16] [35].

Demographic based techniques can be combined and leveraged with other techniques in order to help alleviate such issues as the cold-start problem by requiring a user to create an account with basic demographic information before using the recommendation system. Doing this, the new user cold-start problem is avoided or at least alleviated, as long as there are enough users and ratings in the community already. However, there is a privacy issue due to the sensitivity of some demographic data. This can make the harvesting of all available demographic data for a particular user impossible or unethical [35].

Demographic information can be learned from ways other than explicit user profiles as well. For example, connecting social profiles can provide demographic information to a recommender system for a user. Even before the emergence of social media, simple scraping and classification of text on personal web pages could provide some demographic information for a new user to a system [50].

2.2.7 Computational Intelligence based techniques

Computational intelligence recommender systems include the use of techniques from natural language processing, machine translation and machine learning techniques [20] and more recently Bayesian techniques, neural networks, clustering, genetic algorithms and fuzzy sets [44] in recommender systems. Computational intelligence systems are not mentioned as much in current literature. In more recent literature, often the techniques that are sometimes attributed to computational intelligence recommender systems are grouped into model-based collaborative-filtering recommender systems.

2.2.8 Hybrid recommendation techniques

Hybrid recommender systems typically use a combination of content based and collaborative filtering based techniques [51]. However, increasingly they are also incorporating in more state-of-the-art techniques such as social recommendation techniques and geographical recommendation techniques [16] [41] [68]. Combining techniques from each of these categories can both improve recommendation accuracy and help alleviate some of the problems inherent with each individual model [51]. Many hybrid models still rely on an underlying ratings structure, as was the case for collaborative filtering. Research has been done to eliminate or diminish the reliance on ratings systems for hybrid models by incorporating more sophisticated techniques such as those from machine learning and data mining [68].

Still today, Burke [18] is highly cited for categorizing hybrid recommender systems into seven major categories. These are summarized in Table 2.2.

2.3 Techniques for Recommender Systems Evaluation

Currently, there is no set standard for evaluating a recommender system [67]. However, there are a variety of techniques that can be used to provide different evaluation metrics of a system. Depending on the type and goals of the system, some evaluation metrics may be a better choice than others to benchmark a system. The following section discusses methods to evaluate various aspects of a recommender system.

2.3.1 Decision Support Metrics

Due to the many similarities between recommender systems and search engines, many of the most common evaluation techniques for the two are shared [24] [31] [42] [51] [71]. These are:

1. *Precision*: The number of relevant recommendations returned out of the total number of recommendations returned.

$$P = \frac{N_{rs}}{N_s} \tag{2.8}$$

2. *Recall*: The number of relevant recommendations returned out of the total number of relevant recommendations in the entire collection.

$$R = \frac{N_{rs}}{N_r} \tag{2.9}$$

Table 2.2: Summary of Types of Hybrid Recommendation Schemes

Type	**Description**
Weighted	Several recommendation algorithms are used. A linear combination of their results is taken, given weights, and used to produce the finalized recommendations.
Switching	The recommender system is designed to switch between a variety of different recommendation techniques and algorithms depending on the situation in order to produce the best results.
Mixed	Several recommenders are used at the same time and their results are presented to the user in tandem.
Feature Combination	Combines features from a variety of sources to create a feature combination hybrid and uses them together as input into the recommender system.
Cascade	A series of recommenders are cascaded together with the goal of each subsequent system refining and improving on the results from the previous system.
Feature Augmentation	One recommendation technique is used to produce a rating or classification of an item. Then, the output from this first system is incorporated into the processing steps of the next technique (s).
Meta-level	Uses a model generated by one recommender system as the input for another recommender system.

3. *F1 Measure*: Combined metric of both precision and recall.

$$F1 = \frac{2RP}{R+P} \tag{2.10}$$

where for Equations 2.8 2.9 2.10, P is precision, R is recall, $F1$ is F1 measure, and all other terms are defined as in Table 2.3 [31] [51].

Table 2.3: Confusion Matrix for Recommender Systems

	Selected	**Not Selected**	**Total**
Relevant Recommendation	N_{rs}	N_{rn}	N_r
Irrelevant Recommendation	N_{is}	N_{in}	N_i
Total	N_s	N_n	N

2.3.2 Rank Weighted Indices

Often, the list of potential recommendations for a particular user can be large [16] [45]. When this happens, it may be desirable to evaluate the recommender system by examining where relevant recommendations are positioned in the list of recommendations and where non-relevant recommendations are positioned in the list of recommendations. These rank-weighted measures are increasingly important as a user loses interest the further they must go down a list of recommendations in order to find what is relevant to their needs. Three common techniques are used to evaluate this metric [16] [24] [45]:

1. *Discounted Cumulative Gain*: Measure that gives a higher score to relevant items that are at the top of a top-n recommender list. The decay in score is logarithmic as

the relevant items move further down the list.

$$DCG = \frac{\sum_{i=1}^{p} 2r_i - 1}{\log_2 i} \tag{2.11}$$

where for Equation 2.11, p is $recommendation_i$s position in the top-n list, and r_i is the graded relevance of $recommendation_i$.

2. *Half-Life*: Measure that gives a higher score to relevant items that are at the top of a top-n recommender list. The decay in score is exponential as the relevant items move further down the list.

$$HL_i = \frac{\sum_{\alpha=1}^{N} \max r_{i,\alpha-d,0}}{2^{\frac{o_{i\alpha}-1}{h-1}}} \tag{2.12}$$

where for Equation 2.12, $r_{i\alpha}$ is the recommendation score, $o_{i\alpha}$ is the predicted ranking of object α in the recommendation list of useri, d is the average rating, and h is the rank of the object on the list for which there is a 50% change the user will eventually examine it.

3. *Rank-biased Precision*: Another measure similar to Discounted Cumulative Gain, differing in that decay in score is via a geometric sequence rather than logarithmic.

$$RBP = (1-p)\sum_{i=1}^{L} r_i p^{i-1} \tag{2.13}$$

where for Equation 2.13, r_i is the graded relevance of $recommendation_i$, p is a probability, and N is the length of the recommendation list.

2.3.3 Accuracy Metrics

Accuracy metrics are also commonly used when evaluating recommender systems.

Statistical support accuracy metrics are used to compare the recommender systems predicted ratings with actual user ratings. Three common types of statistical support accuracy are used [24] [31] [42] [51] [68]:

1. *Mean Absolute Error*: Calculates the average difference between the predicted rating and the actual user rating for the entire dataset.

$$MAE = \frac{\sum_{i=1}^{N} |p_i - r_i|}{N} \tag{2.14}$$

2. *Normalized Mean Absolute Error*: Performs the same calculation as Mean Absolute Error, but normalizes the value by dividing by the possible range of ratings. This is useful when comparing Mean Absolute Error values across datasets that have different rating scales.

$$NMAE = \frac{\sum_{i=1}^{N} |p_i - r_i|}{N(r_{high} - r_{low})} \tag{2.15}$$

3. *Mean Root Squared Error*: Calculates the accuracy by giving more weight to predicted values that deviate largely from the user rated values and less weight to predicted values that have a small deviation from the user rated values.

$$RMSE = \frac{\sqrt{\sum_{i=1}^{N} |p_i - r_i|^2}}{N} \tag{2.16}$$

   where for Equations 2.14–2.16, p is the predicted rating, r is the user rating, i refers to user i, and N is the total number of ratings.

2.3.4 Coverage

Coverage of a recommender system is closely related to accuracy metrics [65]. Coverage refers to the percentage of objects that the recommendation system is actually able to recommend to users [31] [45] [65]. This definition of coverage can be represented by:

$$Cov(K) = \frac{N_d}{N} \tag{2.17}$$

where K is the top-K number of recommendations, N_d is the total number of distinct objects in all lists K, and N is the number of unique objects in the system. Low coverage values mean that the recommender can only recommend from a small list of objects in the system whereas a high coverage value indicates it can recommend from a large list of object. Being able to recommend from a large list of objects is desirable as it increases diversity of recommendations [45].

2.3.5 Novelty

Novelty of a recommender system refers to the systems ability to recommend items to a user that the user previously did not know about [31] [45] [65]. There are considered to be two different types of novelty with recommender systems: Popularity-based item novelty and Distance-based item Novelty [60].

Popularity-based item novelty refers to the difference between an item and that item being observed (or discovered) by a user. In a typical recommender system database, novel items are located in the long tail of the database (items that have not been interacted with much, usually a large portion of the database) and non-novel items are located in the head (items that are popular and have been interacted with a great deal). Given this information, novelty of a recommender system can be approximated by the following [60]:

$$nov(i|\theta) = 1 - p(seem|i,\theta) \tag{2.18}$$

where θ represents a variable on which item discovery may depend and p is a conditional probability.

Distance-based item novelty is a non-boolean version of the popularity-based item novelty scheme. Whereas in the popularity scheme, a user has either seen an object or has not, distance schemes calculate novelty by viewing it as a relationship between the item and the context of experience. If it is assumed that $p(j|\theta)$ is of uniform distribution, distance-based item novelty can be approximated by the following [60]:

$$nov(i|\theta) = \frac{\sum_{j\varepsilon\theta}(choose|j,\theta,i)(1-sim(i,j))}{\sum_{j\varepsilon\theta} p(choose|j,\theta,i)} \tag{2.19}$$

where $p(choose|j,\theta,i)$ is the probability that the user chooses item j in context θ given that they chose item i, and sim is a similarity measure (the choice of which can vary).

There are several other definitions on how to calculate novelty throughout the literature [16] [45]. Many of them are very similar to the methods already presented. For example, [16] offers a non-probabilistic way to calculate novelty for a recommender system:

$$nov_i = \frac{1}{K-1}\sum_{j\varepsilon K}|1-sim(i,j)|, i\varepsilon \tag{2.20}$$

where K is the number of items returned in the top-n list, and $sim(i,j)$ is a similarity measure. As can be seen, Equation 2.19 and Equation 2.20 are very similar and differ by a normalization factor.

As there is no set standard for novelty calculations on a recommender system, the exact definition used for finding the novelty of a specific system should be carefully chosen on a case by case basis [16].

2.3.6 Serendipity

Serendipity refers to the surprise factor of recommendations is this recommended item an item of interest to the user that they may not have otherwise discovered [31]. For example, a recommender can give a recommendation that is perfectly relevant and accurate to the user, but if the recommendation is too obvious to the user, then the recommendation loses its merit. A commonly cited example is a grocery recommender system recommending a user to buy bananas. This is an obvious suggestion because almost everyone buys bananas and bananas are placed in an obvious place at the grocery store because retailers already know this.

Mathematically analyzing the serendipity of a recommender system is not an obvious task. One nave suggestion offered by some literature is to calculate the average popularity over the items in the top-n list. A lower score would suggest higher serendipity [71].

2.3.7 Diversity

Diversity in recommender systems refers to how different the given recommendations are from each other within a given top-n list [45]. This metric is important as it has been shown that users tend to be more satisfied with results when there is an appropriate degree of diversity within recommendation lists [39].

One method of measuring the diversity of list of recommendations is through a quantity known as Intra-list Similarity [16] [60] [71]:

$$ILS = \frac{1}{K(K-1)} \sum_{i \varepsilon Z} \sum_{j \varepsilon Z} |1 - sim(i,j)|, i \neq j \tag{2.21}$$

where K is the number of items in the top-n list and *sim* is a similarity measure.

This definition of diversity does not depend on the ordering of the top-n list it only depends on the actual items in the list [71].

2.4 Existing Frameworks for Building Recommendation Engines

A variety of tools are available for implementing and evaluating a recommender system. Although some of these tools require payment, many are free and some are even open source. A more comprehensive list of freely available toolkits can be found in Table 2.4 [29] [37], but a subset of those listed are explained in more detail in the following subsections.

Table 2.4: Free Recommender Frameworks and Toolkits

Software	**Language**	**Description**
Apache Mahout	Java, Scala	Machine learning library includes collaborative filtering
Cofi	Java	Collaborative filtering library
Weka	Java	Machine learning library that can be used for collaborative filtering
easyrec	Java	Recommender for webpages
LensKit	Java	Collaborative filtering algorithms from GroupLens Research
MyMediaLite	C#/Mono	Recommender systems algorithms
PREA	Java	Personalized Recommendation Algorithms Toolkit
recommenderlab	R	Collaborative filtering framework
SVDFeature	C++	Toolkit for feature-based matrix factorization
Vogoo PHP LIB	PHP	Collaborative filtering engine for personalizing websites

2.4.1 easyrec

Easyrec is an open-source toolkit built to allow developers to integrate recommendations into their websites using RESTful services [13]. The toolkit supports both personalized and non personalized recommendations, rankings, manual clustering of items, using item types for type specific recommendations and the use of rule-generators. An administration console is also provided to display usage statistics.

2.4.2 recommenderlab

Recommenderlab is a collaboative filtering algorithm research environment for the statistical programming language R [29]. The primary goal of this package is to support research in the field, not as a primary infrastructure for providing recommender algorithms — although recommendation algorithms can be built using it. This differentiates this framework from many others on the market today. Recommenderlab provides support for developing algorithms and evaluating them with both ratings data or 0–1 (binary) data.

2.4.3 Apache Mahout

Apache Mahout provides an open-source framework for developing machine learning, classification and recommendation algorithms at scale. It includes both a Java API (which will soon be deprecated) and a Scala API. Many pre-defined algorithms are available for collaborative filtering, classification, clustering, dimensionality reduction, topic models, and other miscellaneous tasks for use either via the APIs or scripts that ship with the Mahout download.

To provide for scale, Mahout makes use of Map-Reduce technology with Apache Hadoop (for the Java version), or via Apache Spark, Apache Flink and H2O (for the Scala version) [7].

Chapter 3

Problem

The ACM International Conference on Recommender Systems is an international conference that takes place annually. It has been running since 2007.

In 2010, a corresponding challenge was introduced called the Challenge on Context-aware Movie Recommendation (CAMRa) 2010 challenge. Interested participants were given a real-world problem in recommender systems and challenged to devise a novel or unique solution. Those who were successful submitted a paper to the conference covering information about their approach and the algorithms used. This challenge was so successful that the tradition has been continued [1]. Since 2012, it has been simply called the RecSys challenge.

This project uses the dataset from the RecSys 2016 challenge and compares the approach of various algorithms to solve the given problem. The algorithms that will be used are as follows:

- Content-based (no collaborative filtering)
- Item-based collaborative filtering with Tanimoto Coefficient Similarity
- Item-based collaborative filtering with Log Likelihood Similarity
- User-based collaborative filtering with Tanimoto Coefficient Similarity for each of 2,

16, 128, 1024 Nearest Neighbors

- User-based collaborative filtering with Log Likelihood Similarity for each of 2, 16, 128, 1024 Nearest Neighbors
- Multimodal (Hybrid) Recommender with Log Likelihood Similarity for 512 Nearest Neighbors plus Content Indicators

3.1 Problem Description

The problem is stated as follows by the organizes of the 2016 challenge [2]:

> The task of the challenge is to compute 30 recommendations (or less) for each of the 150,000 target users. In particular, the algorithms have to predict those items that a user will interact with. If you interpret the problem as a ranking problem then you could phrase the task as follows: given a user, the recommender algorithm has to compute a ranked list of items so that those items which the user is more likely to interact with appear at the top of the ranking.

Formally, given a set of users U, where $|U| \geq 150,000$, users u_i such that $u_i \in U$ and $0 \leq i \leq 150,000$, and a set of items I, the problem is to predict an ordered list of the top thirty most likely job postings each user, u_i, will interact with:

$$\forall u \in U, i_u = \underset{i \in I}{\arg\max} P(i_1), P(i_2), ..., P(i_3) \tag{3.1}$$

where the number of items in the list are ≤ 30 and $\forall$ items in the list, the probability of the user interacting with that item is less than or equal to the probability of the the user interacting with the previous one ($P(i_1) \geq P(i_2) \geq P(i_3)$).

3.2 Dataset

The provided dataset is a semi-synthetic sample of data from a German professional social networking site, XING. This site is very similar to the North American professional social networking site, LinkedIn. The dataset is composed of both real XING users and artificial users who were added to help maintain anonymity. Additionally, nearly all user attributes were mapped to integer values without the corresponding reverse mapping functions being made available. This was also done to protect anonymity [2].

Specific anonymization, pseudonymization, and noise techniques applied to the dataset were as follows:

- Addition of artificial users
- Dataset is an incomplete sample of all XING users and job postings
- IDs are used instead of raw text
- Some attributes have been removed
- Some interactions of a user have been removed
- Some interactions in the dataset are artificial
- Timestamps have been shifted

There were four separate datasets that could be utilized in any combination. They are explained in the following subsections.

3.2.1 XING Impressions Dataset

The impressions dataset contains information about job postings that were recommended to users based on the actual recommender system deployed on the XING website. Only a subset of the recommendations were provided; for example, all email recommendations were not provided in the dataset.

This dataset was not used in the project and therefore a more detailed description of this dataset is not provided.

3.2.2 XING Interactions Dataset

The interactions dataset contains information about how users responded to job postings on the website. The data contains both positive (reply, click, bookmark) and negative (delete) interactions, coded by integer values. Information about the time the interaction took place was also available.

A description of the format of this dataset can be found in Table 3.1 [2].

Table 3.1: XING Interactions Dataset Description

Field	**Description**
user_id	A unique ID for a user who performed the interaction
item_id	A unique ID for the item which the interaction was performed on
interaction_type	Integer code mapping to the type of interaction where ”1” = user clicked on item; ”2” = user bookmarked the item; ”3” = user replied to the job posting item; ”4” = user deleted the item
created_at	A timestamp of the time when the interaction took place. In the format of seconds since unix epoch.

3.2.3 XING Users Dataset

The users dataset contains detailed information about the users in the XING dataset that may be useful for making recommendations. Information available about users included information about a user's employment experience, education history, and geographical location. It was possible for any of these fields in the data to be blank, so just because a user was presend in the dataset, it was not guaranteed that information was available for all or even any of the possible fields.

The description of the format of this dataset can be found in Table 3.2 [2].

Table 3.2: XING Users Dataset Description

Field	**Description**
id	A unique ID for a user
jobroles	A comma-separated list of integer ids corresponding to job role terms. These are extracted from a user's job title (s).
career_level	Integer code mapping to the current career level of the user where "0" = unknown; "1" = student or intern; "2" = entry level; "3" = Professional; "4" = Manager; "5" = Executive; and "6" = Senior Executive
discipline_id	An integer ID that maps to the discipline of the user. Examples provided are consulting or human resources.
industry_id	An integer ID that represents the industry the user works in. Examples provided include Internet, Automobile or Finance.

Continuation of Table 3.2

Field	Description
country	A string code representing the country the user is from where “de” = Germany; “at” = Austria; “ch” = Switzerland and “non_dach” = any other country
region	An integer ID mapping to which area in Germany a user lives in, if applicable.
experience_n_entries_class	An integer representing the number of previous entries under “Work Experience” the user has where “0” = no entries; “1” = 1–2 entries; “2” = 3–4 entries; and “3” = 5 or more entries
experience_years_in_current	The estimated number of years of experience the user has at their current job.
edu_degree	An integer ID mapping to the estimated university degree of the user where “0” = unknown; “1” = bachelor; “2” = master; and “3” = phd
edu_fieldofstudies	A comma separated list containing integer IDs mapping to the areas the user studied.

3.2.4 XING Items Dataset

The items dataset contains detailed information about the items in the XING dataset that may be useful for making recommendations. Information that may be present about a job posting include such aspects as job title, industry, career level and geographical location. As with the user dataset, an item with an item id could appear in the dataset with other fields that contained blank or null information.

A detailed description of the format of this dataset can be found in Table 3.3 [2]

Table 3.3: XING Items Dataset Description

Field	**Description**
id	A unique ID for an item
title	A comma-separated list of integer ids corresponding to job role terms. These are extracted from a the title of the job postings.
career_level	Integer code mapping to the career level of the job posting where “0” = unknown; “1” = student or intern; “2” = entry level; “3” = Professional; “4” = Manager; “5” = Executive; and “6” = Senior Executive
discipline_id	An integer ID that maps to the discipline of the job in the posting. Examples provided are consulting or human resources.
industry_id	An integer ID that represents the industry of the job in the posting. Examples provided include Internet, Automobile or Finance.
country	A string code representing the country the user is from where “de” = Germany; “at” = Austria; “ch” = Switzerland and “non_dach” = any other country
region	An integer ID mapping to which area in Germany a user lives in, if applicable.

Continuation of Table 3.3

Field	**Description**
latitude	Latitude information of the location of the job posting. Resolution is to the nearest 10km.
longitude	Longitude information of the location of the job positing. Resolution is to the nearest 10km.
employment	An integer ID mapping to the type of employment being offered where "0" = unknown; "1" = full-time; "2" = part-time; "3" = freelance; "4" = intern; and "5" = voluntary
tags	A comma separated list containing integer IDs mapping to concepts representing tags, skills or company names.
created_at	A time-stamp of the time when the job posting was created. In the format of seconds since UNIX epoch.
active_during_test	An integer that represents whether the job posting is still active and can thus be recommended. A value of "1" indicates it can be recommended while a value of "0" indicates it cannot.

3.2.5 Target Users

Additionally, a subset of 150,000 user ids was provided with which participants are to produce recommendations for. These users are known as the ”Target Users”. In order to use the provided evaluation framework (see Section 3.3 for more details), the users which are ran through the evaluator must be contained in this list. Any users not in this list will not be included in the evaluation.

3.3 Evaluation Framework

Challenge organizers provide the algorithm of the framework with which recommendations will be evaluated. This framework will be referred to as the RecSys Evaluation Framework (REF). This is shown in Algorithm 1.

Based on this algorithm, the score for a particular recommendation algorithm will be calculated based on some held-out golden standard recommendation file. This is unknown to participants but is necessary for the REF to calculate the score in the method described. The heart of the algorithm is described by Equation 3.2:

$$\begin{aligned} Score() = {} & 20 * (P@2(r,t,2) + P@4(r,t,4) + r(r,t) + \\ & userSuccess(r,t)) + \\ & 10 * (P@6(r,t,6) + P@20(r,t,20) \end{aligned} \tag{3.2}$$

where r is an ordered list of recommendations for user u, t is the set of relevant items for user u based on the gold standard.

The algorithm for REF is based on three sub-functions. These are shown in Algorithms 1–4.

Algorithm 1 shows the main REF function. This function calls each of the subfunctions necessary to calculate the final score as was defined by Equation 3.2. First, in lines 5-8, the sum of precision@2, precision@4, recall and userSuccess is calculated for all target users. Then, in line 9, the sum of these scores is multiplied by 20. This has the effect of boosting the score for recommendations with precision in the top 2 and 4. Next, also in line 9, the sum of precision@6 and precision@20 is calculated. This sum is then multiplied by 10 and added to the previous sum. In line 11, the final score is then returned to the calling function.

Algorithm 1 Recsys Evaluation Framework Main Score Function

```
procedure SCORE($S$, $T$)
    $score = 0.0$
    for all $u,t \in T$ do
        $r \leftarrow S(u)$
        $score \leftarrow score$ + precisionAtK ($r$, $t$, 2)
        $score \leftarrow score$ + precisionAtK ($r$, $t$, 4)
        $score \leftarrow score$ + recall ($r$, $t$)
        $score \leftarrow score$ + userSuccess ($r$, $t$)
        $score \leftarrow$20 * $score$ + 10 * (precisionAtK ($r$, $t$, 6) + precisionAtK ($r$, $t$, 20))
    end for
    return $score$
end procedure
```

Next, Algorithm 2 shows the sub-function that is responsible for calculating the precision portion of the REF score. This function simply gets the top k recommendations for a given user and sees how many are in common with the known golden recommendations. This procedure is straightforward in lines 2 - 3.

Algorithm 2 Recsys Evaluation Framework Precision Function

```
1: procedure PRECISIONATK($recommendedItems$, $relevantItems$, $k$)
2:     topK $\leftarrow$ $recommendedItems$.take($k$)
3:     return intersect ($topK$, $relevantItems$).size / $k$
4: end procedure
```

Algorithm 3 provides functionality for calculating the recall portion of the score in the evaluation framework. First, in line 1, the algorithm checks if there are any relevant items available to recommend. If there is, then we continue. If there isn't, then given the definition of recall in Equation 2.9, there is no need for the function to continue and the value 0.0 is returned via line 6. In line 3, up to 30 items are recommended for the user, and those item ids are then intersected with the known golden relevant items to find how many are in common. This value is then returned in line 4.

Algorithm 3 Recsys Evaluation Framework Recall Function

```
1: procedure RECALL($recommendedItems$, $relevantItems$)
2:     if $relevantItems$.size $> 0$ then
3:         $interesection\_size$ $\leftarrow$ intersect ($recommendedItems$.take (30), $relevantItems$).size
4:         return $intersection\_size$
5:     else
6:         return 0.0
7:     end if
8: end procedure
```

Finally, Algorithm 4 is the last sub-function of the evalucation framework. This calculates a quantity called ”User Success”, a method of attempting to quantify how likely it was that the recommendations make the experience a success for the user by creating good quality recommendations. The algorithm scores this by first determining, in line 2, whether or not there are any relevant items in the top 30 recommendations. If there are none, then the user success score is determined to be 0, and the value 0.0 is returned via line 5. Otherwise, a score of 1 is given, and a value of 1.0 is returned via line 3.

Algorithm 4 Recsys Evaluation Framework UserSuccess Function

```
1: procedure UserSuccess($recommendedItems$, $relevantItems$)
2:     if intersect ($recommendedItems$.take (30), $relevantItems$) $> 0$ then
3:         return 1.0
4:     else
5:         return 0.0
6:     end if
7: end procedure
```

Chapter 4

Methodology

This chapter will describe the general approach used to solve the problem described in Chapter 3. First, the environment used to develop the solution will be described. Then, the technologies used to develop the solution will be briefly discussed. Finally, the major technology used to develop the solution, Mahout, along with the algorithms used on this technology will be detailed.

4.1 Development Environment

The environment used for the development of this project was a Mid 2015 MacBook Pro. The specifications of this system were:

- **Operating System**: MacOS Sierra (version 10.12.1)
- **Processor**: 2.2 GHz Intel Core i7
- **Memory**: 16 GB 1600 MHz DDR3
- **Disk**: 500 GB SSD
- **Graphics**: Intel Iris Pro 1536 MB

4.2 Technologies

Several pre-existing technologies were leveraged in the implementation of the recommender engine to solve the given problem. All solutions used were freely available for download on the Internet; many were open-source projects.

4.2.1 Apache Solr

Apache Solr is an open-source reliable, scalable, and fault-tolerant search engine built on Apache Lucene. Documents are indexed into Solr over HTTP via JSON, XML, CSV or binary. These documents can then be retrieved by queries through a RESTful service via JSON, XML, CSV, or binary [10]. Solr can be run on one machine as a single-node service, or in distributed mode (known as SolrCloud) on a single or multiple machines.

For this project, Solr was utilized as a database to store data about users and job postings according to a defined schema. A diagram showing the detailed schema is shown in Figure 4.1. The data fields and their parameters are as follows:

- *id*: Primary key for all tables. Field parameters are string type, single valued, indexed, and stored.

- *orig_user_id*: The userID as defined in the original data files. Field parameters are integer type, single valued, indexed, and stored.

- *user_id*: Remapped userId from 0-n without any gaps, required for proper Mahout function. Field parameters are integer type, single valued, indexed, and stored.

- *item_id*: The itemID as defined in the original data files. Field parameters are integer type, single valued, indexed, and stored.

- *geo*: Location field. Field parameters are location type, single valued, indexed, and stored.

- *interaction_type*: Value indicating whether the interaction is a click, bookmark, reply, or delete. Field parameters are integer type, single valued, indexed, and stored.
- *active_during_test*: Indicates whether the item is valid for recommendation. Field parameters are integer type, single valued, indexed, and stored.
- *tags*: Tags that indicate aspects about items. In the 'Users' table, tags represent the tags of items that the user has clicked on. Field parameters are text_general type, single valued, indexed, and stored.
- *user_click_similarity*: ItemIDs of items that are similar to items that this user has clicked on. Calculated by the mahout spark-itemsimilarity job. Field parameters are text_general type, single valued, indexed, and stored.
- *user_click_similarity*: ItemIDs of items that the user is likely to bookmark or reply to after clicking on. Calculated by the mahout spark-itemsimilarity job. Field parameters are text_general type, single valued, indexed, and stored.

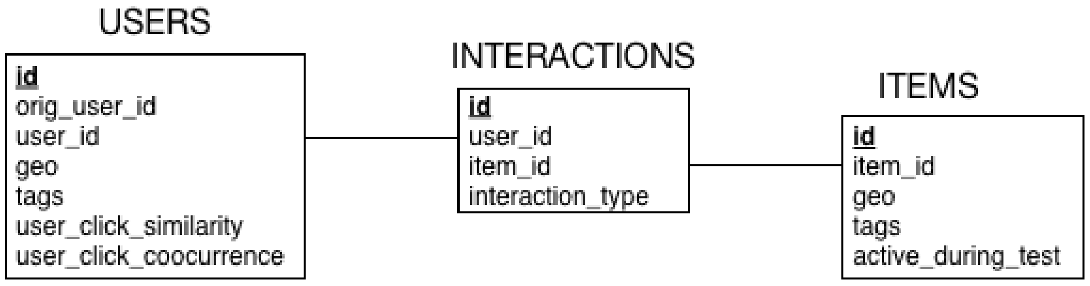


Figure 4.1: Solr schema diagram

At the time of development, the most recent release of Solr was used, version 6.2.0. Because of the relatively low query load expected on Solr while running the recommender for this project, Solr was configured to run as a single node on localhost. In order to pull large amounts of data from Solr in one query, the JVM heap space was modified to have a maximum allocation of 4GB.

4.2.2 Apache Spark

Apache Spark is an open-source, large-scale, general-purpose data processing engine. Data can be processed up to one-hundred times faster on a Spark cluster than on a more traditional Apache Hadoop cluster [8]. It supports multiple programming languages including Java, Scala, Python, and R.

For this project, the most current version of Spark was version 2.11.8 with default configuration settings were used.

4.2.3 Apache Mahout

Apache Mahout is a recommendation framework available in both Java and Scala. It is known for it's variety of included collaborative filtering algorithms that have the ability to perform well out of the box.

At the time of development, the most recent version of Apache Mahout, version 0.12.2, was utilized. For this project, Mahout was used both to implement trivial collaborative filtering recommenders, and to implement a more complex multimodal recommender through integration with Solr and Spark. In order the process the large amounts of data and integrate with Solr and Spark, the default JVM size configuration was increased to 6 GB. Otherwise, default configuration values were used.

4.2.4 Python

A large part of the project involved the cleaning and processing of various data files both provided by the datasets and created from Apache Spark as well as the ingestion of the cleaned data into Solr for storage. To perform these tasks, a script was written using Python 2.7.12 and the open-source text editor, Atom. A copy of relevant portions of this data cleaning and Solr ingestion script can be found in Appendix A.

4.2.5 Java

Code for implementation of the pure collaborative-filtering based recommendation systems and for generating the output of the multimodal recommender were written using Oracle Java 1.8.0.91 in the IntelliJ 2016 Community Edition IDE. Several additional libraries were used in the implementations which are described in the following Subsection 4.2.5.

Key components of the Java code for this project are provided in Appendix B.

Additional Utilities Used

Several additional Java libraries were utilized to ease development. These included:

1. *Apache Maven*: An open-source dependency management framework for Java. Used for loading and managing external JARs required by the implementation [9].

2. *Apache SolrJ*: A Java library to easily access, query, and update Solr indexes [59].

3. *FastUtil*: A Java library developed by University of Milano that extends the Java Collections Framework. Provides more computationally and memory efficient implementations of common data structures such as hash maps [61].

4. *Apache Commons CSV*: A Java library for reading and writing CSV files [6].

4.3 Approach

Due to the accepted good performance of collaborative filtering algorithms, a focus on collaborative filtering algorithms was chosen for this project. Because of this choice, Apache Mahout was the framework chosen to build the project upon. Mahout contains a well-implemented and well-tested variety of collabative filtering algorithms. Although there are recommendation frameworks that claim to have better performance than Mahout, the Mahout framework has decent performance on small to midsize datasets, like that used for this project.

There are two major types of collaborative filtering algorithms that Mahout is well-equipped to deal with: item-based and user-based. Both methods have pros and cons in regards to the problem solved by this project. These will be discussed in Section 4.3.2 for item-based collaborative filtering and Section 4.3.2 for user-based collaborative filtering. Collaborative filtering algorithms can also be combined with another recommendation technique, most commonly a content based technique, to create a hybrid recommender (typically called a multimodal recommender in Mahout vernacular). This type of recommender also has pros and cons for the problem approached by this project, as will be discussed in Section 4.3.3.

Each of the three types of collaborative filtering recommenders will be implemented on the project dataset and compared to an additional content-based baseline algorithm described in Section 4.3.1.

4.3.1 Content-based Recommendation Algorithm

Baseline Algorithm

A baseline algorithm was provided as a starting point for participants. This baseline algorithm achieves a score of 26857.38 out of a possible 15,000,000 from the evaluation algorithm (see Section 3.3 for details). The algorithm used to achieve this score was a content-based approach using no other techniques. High-level pseudocode for the algorithm can be found in Algorithm 5 - 6.

The Baseline algorithm works as follows: First, in line 2, the algorithm finds all items that are valid for recommendation by looking at the flag *active_during_test*. From those that are valid, line 3 chooses the top 100 job title based and stores them. In lines 5 – 19, the process is repeated for several different content criteria. Next, in line 20, the scores from the different content recommendations are aggregated. Finally, in lines 21 – 28 we remove any items where the career level of the job posting does not agree with that of the user (assuming that if the user has not actually indicated their career level, it is level 3).

Algorithm 5 Baseline Algorithm (Part 1)

```
1: procedure BASELINEALGORITHM
2:     for all $items$ where $active\_during\_test$ == 1 do
3:         $job-title-based$[] ← top 100 items where $users$.jobroles == $items$.title
4:     end for
5:     for all $item_i \in job-title-based$ do
6:         $score_i$ ← number of overlapping IDs * 3
7:     end for
8:     $job\_tag\_based$[] ← top 100 items where $users$.jobroles == $items$.tags
9:     for all $item_j \in job\_tag\_based$ do
10:        $score_j$ ← number of overlapping IDs * 2
11:    end for
12:    $discipline\_and\_region\_based$[] ← 100 random items where $users$.discipline_id ==
   $items$.discipline_id && $users$.region == $items$.region
13:    for $item_k \in discipline\_and\_region\_based$ do
14:        $score_k \leftarrow 2$
15:    end for
16:    $industry\_and\_region\_based$[] ← 100 random items where $users$.industry_id ==
   $items$.industry_id && $users$.region == $items$.region
17:    for all $item_l in industry\_and\_region\_based$ do
18:        $score_l \leftarrow 1$
19:    end for
20:    Aggregate scores for each item
```

Algorithm 6 Baseline Algorithm (Part 2)

```
21:     for all $item_i \in items$ do
22:         if $users$.career_level != $items$.career_level then
23:             Remove $item_i$ from consideration
24:         end if
25:         if $users$.career_level == $NULL$ —— $users$.career_level == 0 then
26:             $users$.career_level ← 3
27:         end if
28:     end for
29: end procedure
```

4.3.2 Collaborative Filtering Recommendation Algorithms

Item-based Collaborative Filtering

Item-based collaborative filtering algorithms look at the items (in this case, jobs) that a user has rated (or interacted with), and uses that information to decide how similar two users, *user_i* and $user_j$ are. Using this information, the algorithm can then infer for *user_i*, which additional items *user_j* has rated highly that *user_i* is also likely to rate highly. This is illustrated in Algorithm 7. The similarity measure referred to in line 4 can be any similarity measure. Some common measures were discussed in Section 2.1.4.

For this project, the dataset we are using is interaction based rather than ratings based, and thus is boolean in nature. Therefore, the algorithm will be run with two different similarity measures that are known to perform well with boolean data. These are Tanimoto Coefficients and Log Likelihood. The equations are reprinted below for convenience in Equation 4.1 and Equation 4.2, respectively:

$$T(r_{i,u}, r_{j,u}) = \frac{r_{i,u} \cap r_{i,j}}{(r_{i,u} \cup r_{i,j})} \qquad (4.1)$$

Algorithm 7 Item-based collaborative filtering

```
1: procedure ITEM-BASED COLLABORATIVE FILTERING
2:     for all $item_i$ which $user_u$ has no preference do
3:         for all $user_v$ which has a preference for $item_i$ do
4:             Compute similarity s between $user_u$ and $user_v$
5:         end for
6:         Add $user_v$'s preference for $item_i$, weighted by s, to a running average
7:     end for
8:     return top items, ranked by weighted average
9: end procedure
```

$$\lambda = \frac{\max_{w\varepsilon\Omega_0} H(\omega;k)}{\max_{w\varepsilon\Omega} H(\omega;k)} \tag{4.2}$$

Item-based collaborative filtering typically is less computationally intensive than user-based collaborative filtering. This makes it ideal if recommendations are to be performed in real time rather than precomputed ahead of time and uploaded to a server at set intervals. However, it tends to produce poorer quality recommendations in regards to quality metrics such as precision and recall.

User-based Collaborative Filtering

User-based collaborative filtering algorithms attempt to find users similar to a user, $user_u$, by comparing user rating history, and use these users to predict items that *user_u* is likely to rate highly. Algorithm 8 shows pseudocode of a generic user-based collaborative filtering algorithm. As in item-based collaborative filtering, the similarity measure referred to in line 4 can be any similarity measure. Some common measures were discussed in Section 2.1.4.

Once again, for this project, the algorithm will be run with two similarity measures: Tanimoto Coefficients and Log Likelihood. Additionally, the algorithm will be run with a variable number of neighbors, or how many number of users *user_u*, will be compared

Algorithm 8 User-based collaborative filtering

```
1: procedure USER-BASED COLLABORATIVE FILTERING
2:     for all $item_i$ which $user_u$ has no preference do
3:         for all $item_j$ that $user_u$ has a preference do
4:             Compute a similarity s between $item_i$ and $item_j$
5:         end for
6:         Add $user_u$'s preference for $item_j$, weighted by s, to a running average
7:     end for
8:     return top items, ranked by weighted average
9: end procedure
```

with. For this project we will run each similarity measure with neighbors set to 2, 16, 128, and 1024 neighbors.

User-based collaboritve filtering algorithms tend to perform very well in regards to metrics such as precision and recall. However, they are computationally intensive and thus, do not scale well. If a system wants to recompute recommendations in real time, this can result in significant latency for a user. Additionally, if a dataset is very sparse, that is, contains many items for which very few or no users have provided ratings for, these items will not show up in recommendations. Because of this, some users may recieve none or poor quality recommendations.

Collaborative Filtering Architecture

The architecture for running collaborative filtering algorithms in Mahout is shown in Figure 4.2. The main components required are: (1) a datastore to store user preference information; this project uses Solr for this; (2) users with preference information; (3) a recommender datastucture with corresponding helper datastructures such as a neighborhood and preference inferrer; and (4) an application where the recommendations are used.

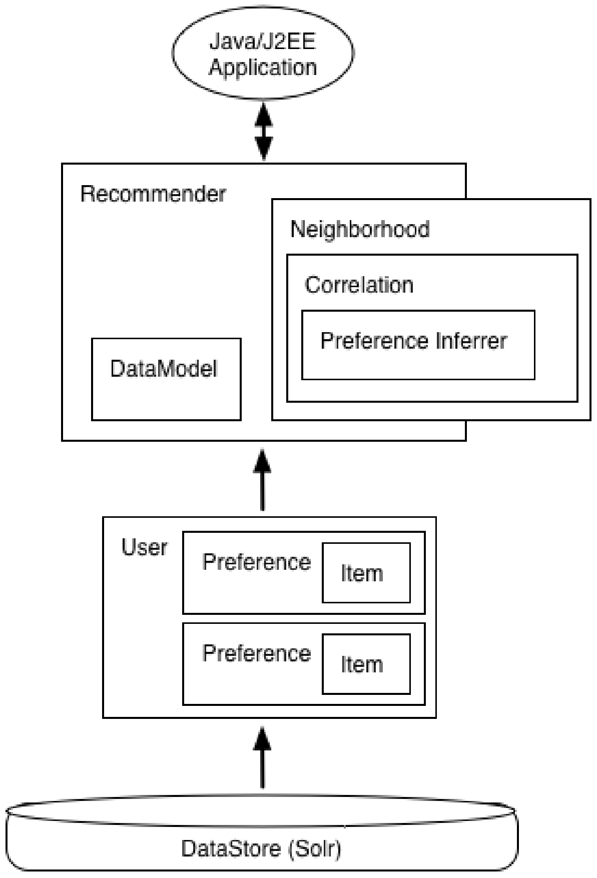


Figure 4.2: Diagram of Apache Mahout collaborative filtering architecture

4.3.3 Multimodal Recommendation Algorithm

A multimodal recommender is a type of hybrid recommender. It is also sometimes called a coocurrence recommender since it often makes use of coocurrence data such as information about how a user clicking on job posting may lead to a user replying to or bookmarking a job posting. Although this coocurrence information is typically calculated through collaborative filtering with log likelihood similarity, it is often combined with other indicators [10]. For example, it can be combined with content tags or geographical location. This combination of collaborative filtering data with content data is what makes a multimodal recommender a type of hybrid recommender.

Pseudocode for the creating the multimodal recommender used for this project is found in Algorithm 9. In line 2 - 3, spark-itemsimilarity refers to the task available in Mahout. For this project, the primary action chosen for the similarity matrix was clicking on a job posting while the cooccurence actions were chosen to be bookmarking or replying to a job posting after clicking on it. The tags in line 5 were found by saving to each user, the tags of each item the user clicked on. This was all then saved in Solr as line 7 states.

When the model is created, an application is ready to use it to make recommendations. This is done by simply creating a disjunctive query over the indicator fields that one wishes to query. In the example Algorithm, this would mean a disjunctive query over the similarity field, coocurrence field, and tags field for a user. The query would return back a list of item ids; these item ids should be the ones that the user is likely to prefer.

Multimodal Recommender Architecture

The architecture for the multimodal recommender is shown in Figure 4.3. The diagram shows there are two clear, and distinct sides: a realtime side the runs the application, and a background side that serves to handle model creation and maintenance. A database is required to store user data and the search engine indexes model data once it is created.

For this project, the database and the search engine portion are both serviced by an

Algorithm 9 Multimodal recommendation Algorithm

```
1: procedure MULTIMODAL RECOMMENDER($I$, $U$)
2:     Calculate the similarity matrix with spark-itemsimilarity
3:     Calculate the coocurrence matrix with spark-itemsimilarity
4:     for all users in $U$ do
5:         Find the tags corresponding to the primary user action
6:     end for
7:     Store all data in a database
8: end procedure
```

Apache Solr instance. However, the database can be another technology such as MySQL or Apache Cassandra.

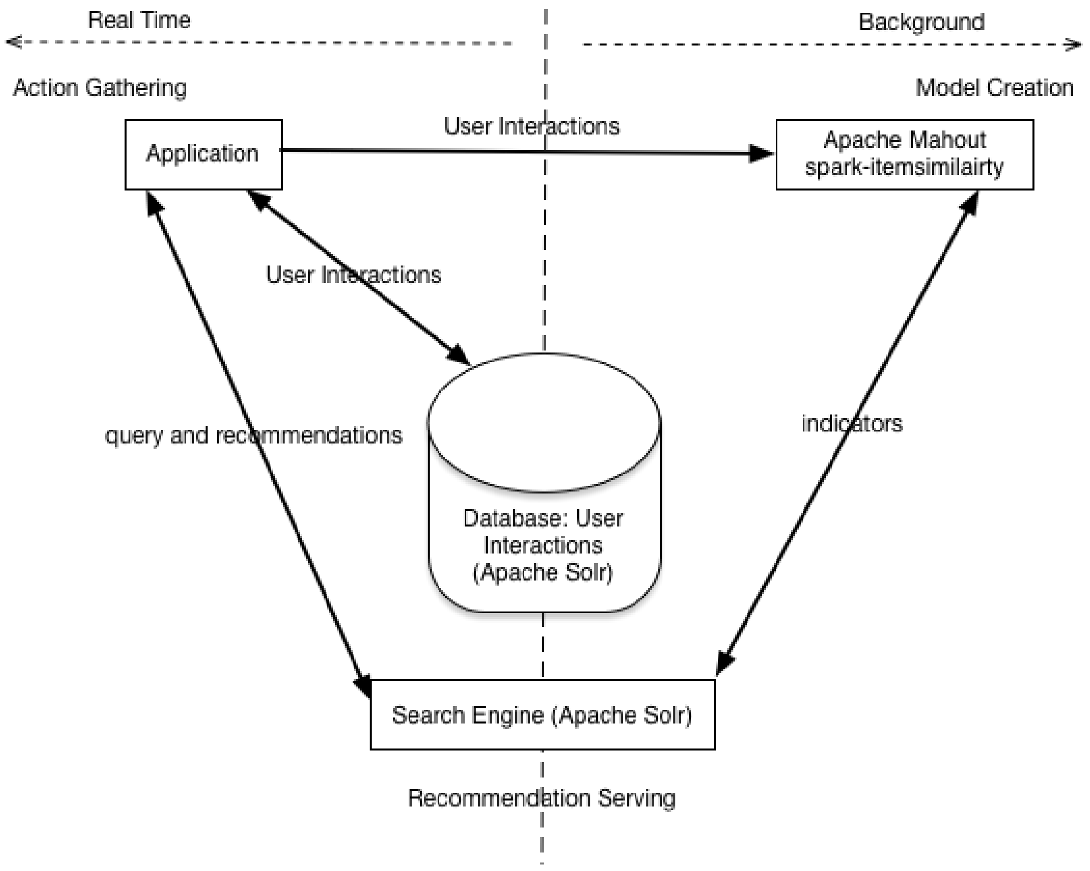

Real Time
Background
Action Gathering
Model Creation
Application
User Interactions
Apache Mahout spark-itemsimilairty
User Interactions
query and recommendations
indicators
Database: User Interactions (Apache Solr)
Search Engine (Apache Solr)
Recommendation Serving

Figure 4.3: Diagram of architecture for a multimodal recommender using Apache Mahout and Apache Solr

Chapter 5

Results

The results of this project largely proved the hypothesis that collaborative filtering recommenders perform better than content-only recommendation systems. Unfortunately, the implementation of the multimodal recommender for this project failed to meet expectations. This is likely due to implementation issues that need to be studied further.

A more detailed discussion of the results follow.

5.1 Discussion

A summary of the numerical scores obtained from the evaluation system are shown in Table 5.1. The first column in the table gives a brief descriptive name of the algorithm while the second column in the table gives the score achieved by the algorithm outlined in Section 4.3.1.

5.2 Collaborative Filtering Results versus Baseline Algorithm Results

As the table shows, collaborative filtering systems performed better than the baseline, content-based, system. As expected, the greater number of neighbors considered in the calculation improved the results of the system; however, the gain in score diminishes between the 128 and 1024 neighbor mark. Since adding neighbors increases the complexity of the calculation, it was unnecessarily time consuming to study neighborhoods greater than 1024 as the gain in score would continue to become smaller in comparison to the computation gain. The exact relationship between computation and neighborhood size was not studied in this project.

Both similarity algorithms performed well. Tanimoto Coefficients performed better when neighborhoods were smaller in regards to score; however, once neighborhood size reached 1024, both algorithms performed equally well. A figure is provided to help illustrate the difference in the performance of these two similarity measures. Figure 5.1 shows the plot for the evaluation score versus the neighborhood size for the Tanimoto Coefficient similarity measure while Figure 5.2 shows the same plot for Log Likelihood similaity. Finally, Figure 5.3 shows a comparison between the two.

It was observed that in regards to computational efficiency, Log Likelihood did run much faster. However, once again, this was not formally studied. It would be an interesting area for future study.

5.2.1 Multimodal Recommender Results

The results for the multimodal recommender clearly indicate that something in the implementatation was likely done incorrectly due to the score being so much lower than even the baseline algorithm. Upon inspection of the recommender, there are two critical areas where there most likely there may have been error introduced. These are: (1) incorrect filtering of job postings that are no longer avaialable for recommendation; (2) potential mistakes in Solr query. Issue number (1) may seem like a minor influencer on the surface, however, when there is a database of many millions of job postings where only several thousand are eligible for recommendation, incorrect filtering of which postings can and cannot be recommended can be detrimental to a recommendation system.

Table 5.1: Summary of REF Score Values for Different Algorithms

Algorithm	**Score**
Baseline Algorithm	26857.38
Item-based CF with Tanimoto Coefficients	47431.89
Item-based CF with Log Likelihood	63410.20
User-based CF with Tanimoto Coefficients and 2 Neighbors	39826.73
User-based CF with Tanimoto Coefficients and 16 Neighbors	96898.34
User-based CF with Tanimoto Coefficients and 128 Neighbors	155814.10
User-based CF with Tanimoto Coefficients and 1024 Neighbors	160501.77
User-based CF with Log Likelihood and 2 Neighbors	32515.35
User-based CF with Log Likelihood and 16 Neighbors	62060.51
User-based CF with Log Likelihood and 128 Neighbors	131625.72
User-based CF with Log Likelihood and 1024 Neighbors	161159.30
Multimodal Recommender	827.59

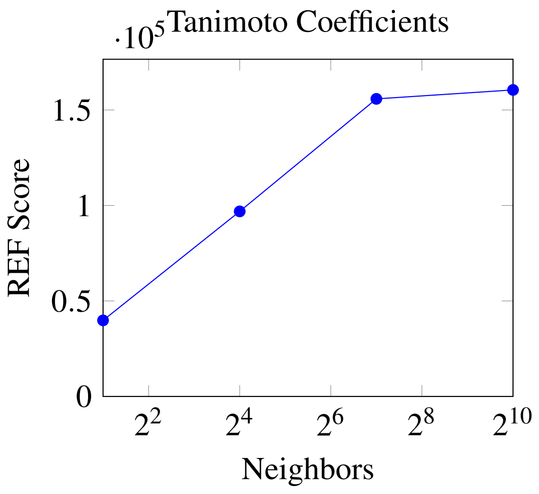


Figure 5.1: User-based CF for TC

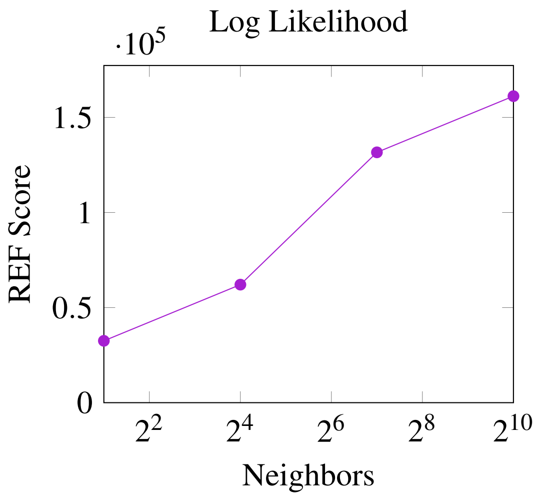


Figure 5.2: User-based CF for LLH

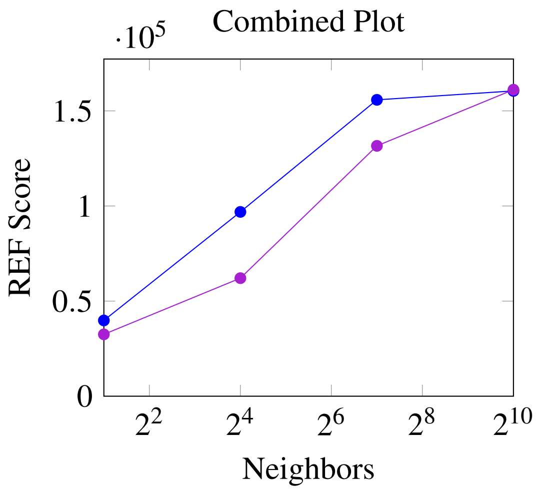


Figure 5.3: Comparison of REF Scores for User-based CF

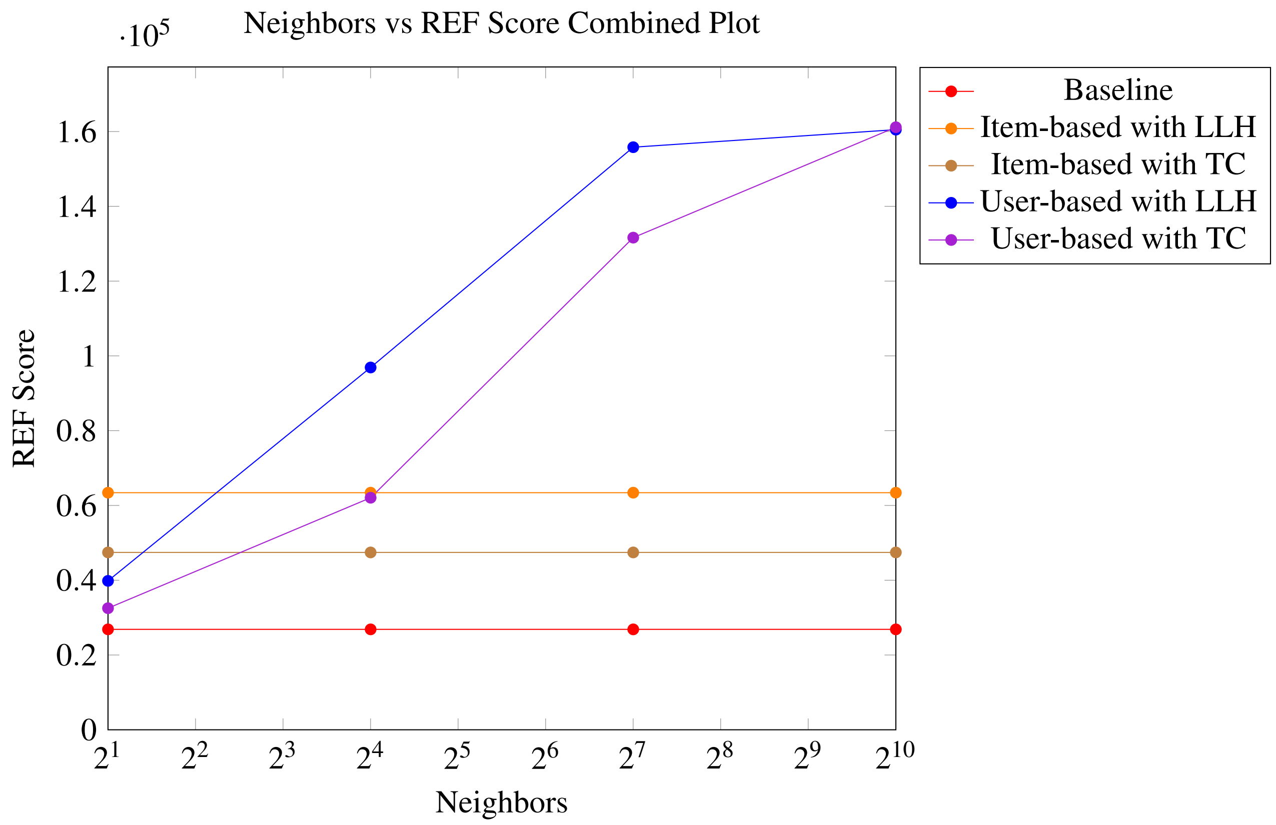


Figure 5.4: Recsys Evaluation Framework Score combined plot

Chapter 6

Conclusion

This chapter serves to summarize the final results of this project as well as provide potential avenues for future work.

6.1 Future Work

Multimodal recommenders have become popular relatively recently. Due to the large amount of data now available, and the large amount of data available to be collected from users, multimodal recommenders have the potential to be truly tuned to the business needs of a recommender.

Future work for this project would involve further refinement of the multimodal recommender. First, the current issues with the multimodal recommender will need to be debugged and resolved. After this work is complete, there are many avenues for further improvement of the system. For instance, currently, geographical information is only used as a fallback for users who do now have tag and click similarity or click coocurrence information. One of the first steps would be to integrate geographical information into the recommender for all users since it is likely that many users have preference for job postings located in certain geographical locations.

A second area for future work would involve a study of the computational complexity of different recommendation algorithms and/or recommendation schemes. Because this project used offline evaluation, computation speed was not importation and thus was not evaluated. However, in a real-world recommender implementation, a user does not want to have to wait a long period of time to see recommendations. A study of the computational complexity of recommendation algorithms, particularly of how adding and removing different factors such as geographical location, demographic information, etc., effects the computation time would be interesting and add value to the field of recommendation systems.

6.2 Concluding Remarks

Although the multimodal recommender in this project did not perform better than the other recommendation algorithms, this was potentially due to implementation issues and not an issue with the recommender itself. Therefore, further study has to be done to determine whether or not the multimodal recommender can, in fact, perform superiorly to a collaborative filtering recommender for job recommendation. According to the current literature and websites such as LinkedIn and XING, it should be able to do so when implemented correctly.

Bibliography

[1] Recommender systems challenge - recsyswiki. `http://www.recsyswiki.com/wiki/Recommender_Systems_Challenge`, 2016. (Accessed on 05/09/2016).

[2] Fabian Abel. recsyschallenge/2016. `https://github.com/recsyschallenge/201`, 2016. (Accessed on 05/09/2016).

[3] Gregory D Abowd, Anind K Dey, Peter J Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a better understanding of context and context-awareness. In *Handheld and ubiquitous computing*, pages 304–307. Springer, 1999.

[4] Panagiotis Adamopoulos and Alexander Tuzhilin. On over-specialization and concentration bias of recommendations: Probabilistic neighborhood selection in collaborative filtering systems. In *Proceedings of the 8th ACM Conference on Recommender systems*, pages 153–160. ACM, 2014.

[5] Gediminas Adomavicius and Alexander Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *Knowledge and Data Engineering, IEEE Transactions on*, 17(6):734–749, 2005.

[6] Apache. Apache commons csv. `https://commons.apache.org/proper/commons-csv/`, 2016. (Accessed on 11/25/2016).

[7] Apache. Apache mahout. `http://mahout.apache.org/`, 2016. (Accessed on 11/25/2016).

[8] Apache. Apache spark. `http://spark.apache.org/`, 2016. (Accessed on 11/25/2016).

[9] Apache. Apache maven. `https://maven.apache.org/`, 2016. (Accessed on 11/25/2016).

[10] Apache. Apache solr. `http://lucene.apache.org/solr/`, 2016. (Accessed on 11/25/2016).

[11] Nana Yaw Asabere. Towards a viewpoint of context-aware recommender systems (cars) and services. *International Journal of Computer Science and Telecommunications*, 4(1):10–29, 2013.

[12] Ebunoluwa Ashley-Dejo, Seleman Ngwira, and Tranos Zuva. A survey of context-aware recommender system and services. In *Computing, Communication and Security (ICCCS), 2015 International Conference on*, pages 1–6. IEEE, 2015.

[13] Research Studios Austria. easyrec :: open source recommendation engine. `http://easyrec.org/`, 2016. Accessed on 05/14/2016.

[14] Gilbert Badaro, Hazem Hajj, Wassim El-Hajj, and Lama Nachman. A hybrid approach with collaborative filtering for recommender systems. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2013 9th International*, pages 349–354. IEEE, 2013.

[15] Lucas Bernardi, Jaap Kamps, Julia Kiseleva, and Melanie JI Mueller. The continuous cold start problem in e-commerce recommender systems. *arXiv preprint arXiv:1508.01177*, 2015.

[16] Jesús Bobadilla, Fernando Ortega, Antonio Hernando, and Abraham Gutiérrez. Recommender systems survey. *Knowledge-Based Systems*, 46:109–132, 2013.

[17] Jesus Bobadilla and Francisco Serradilla. The effect of sparsity on collaborative filtering metrics. In *Proceedings of the Twentieth Australasian Conference on Australasian Database-Volume 92*, pages 9–18. Australian Computer Society, Inc., 2009.

[18] Robin Burke. Hybrid recommender systems: Survey and experiments. *User modeling and user-adapted interaction*, 12(4):331–370, 2002.

[19] Robin Burke, Michael P Mahony, and Neil J Hurley. Robust collaborative recommendation. In *Recommender Systems Handbook*, pages 961–995. Springer, 2015.

[20] Nick Cercone, Lijun Hou, Vlado Keselj, Aijun An, Kanlaya Naruedomkul, and Xiaohua Hu. From computational intelligence to web intelligence. *Computer*, 35(11):72–76, 2002.

[21] Edjalma Q da Silva, Camilo Junior, G Celso, Luiz Mario L Pascoal, and Thierson C Rosa. An evolutionary approach for combining results of recommender systems techniques based on collaborative filtering. In *Evolutionary Computation (CEC), 2014 IEEE Congress on*, pages 959–966. IEEE, 2014.

[22] Ted Dunning. Accurate methods for the statistics of surprise and coincidence. *Computational linguistics*, 19(1):61–74, 1993.

[23] Michael D Ekstrand, F Maxwell Harper, Martijn C Willemsen, and Joseph A Konstan. User perception of differences in recommender algorithms. In *Proceedings of the 8th ACM Conference on Recommender systems*, pages 161–168. ACM, 2014.

[24] Michael D Ekstrand, John T Riedl, and Joseph A Konstan. Collaborative filtering recommender systems. *Foundations and Trends in Human-Computer Interaction*, 4(2):81–173, 2011.

[25] Asmaa Elbadrawy and George Karypis. User-specific feature-based similarity models for top-n recommendation of new items. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 6(3):33, 2015.

[26] Baruch Fischhoff. Value elicitation: Is there anything in there? *American psychologist*, 46(8):835, 1991.

[27] Alex G. The nine must-have datasets for investigating recommender systems, February 2016 (accessed May 3, 2016).

[28] Rainer Gemulla, Erik Nijkamp, Peter J Haas, and Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 69–77. ACM, 2011.

[29] Michael Hahsler. recommenderlab: A framework for developing and testing recommendation algorithms. *Nov*, 2011.

[30] Negar Hariri, Bamshad Mobasher, and Robin Burke. Query-driven context aware recommendation. In *Proceedings of the 7th ACM conference on Recommender systems*, pages 9–16. ACM, 2013.

[31] Jonathan L Herlocker, Joseph A Konstan, Loren G Terveen, and John T Riedl. Evaluating collaborative filtering recommender systems. *ACM Transactions on Information Systems (TOIS)*, 22(1):5–53, 2004.

[32] Natasa Hoic-Bozic, Martina Holenko Dlab, and Vedran Mornar. Recommender system and web 2.0 tools to enhance a blended learning model. 2015.

[33] Mehdi Hosseinzadeh Aghdam, Negar Hariri, Bamshad Mobasher, and Robin Burke. Adapting recommendations to contextual changes using hierarchical hidden markov

models. In *Proceedings of the 9th ACM Conference on Recommender Systems*, pages 241–244. ACM, 2015.

[34] Chia-Ling Hsiao, Zih-Syuan Wang, and Wei-Guang Teng. An incremental scheme for large-scale social-based recommender systems. In *Data Science and Advanced Analytics (DSAA), 2014 International Conference on*, pages 128–134. IEEE, 2014.

[35] Sarika Jain, Anjali Grover, Praveen Singh Thakur, and Sourabh Kumar Choudhary. Trends, problems and solutions of recommender system. In *Computing, Communication & Automation (ICCCA), 2015 International Conference on*, pages 955–958. IEEE, 2015.

[36] Arjan JP Jeckmans, Michael Beye, Zekeriya Erkin, Pieter Hartel, Reginald L Lagendijk, and Qiang Tang. Privacy in recommender systems. In *Social media retrieval*, pages 263–281. Springer, 2013.

[37] Tim Jones. Recommender systems, part 2: Introducing open source engines. `http://www.ibm.com/developerworks/library/os-recommender2/#resources`, December 2013. (Accessed on 05/14/2016).

[38] Bart P Knijnenburg, Martijn C Willemsen, Zeno Gantner, Hakan Soncu, and Chris Newell. Explaining the user experience of recommender systems. *User Modeling and User-Adapted Interaction*, 22(4-5):441–504, 2012.

[39] Joseph A Konstan and John Riedl. Recommender systems: from algorithms to user experience. *User Modeling and User-Adapted Interaction*, 22(1-2):101–123, 2012.

[40] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, (8):30–37, 2009.

[41] Pigi Kouki, Shobeir Fakhraei, James Foulds, Magdalini Eirinaki, and Lise Getoor. Hyper: A flexible and extensible probabilistic framework for hybrid recommender

systems. In *Proceedings of the 9th ACM Conference on Recommender Systems*, pages 99–106. ACM, 2015.

[42] Masupha Lerato, Omobayo A Esan, Ashley-Dejo Ebunoluwa, SM Ngwira, and Tranos Zuva. A survey of recommender system feedback techniques, comparison and evaluation metrics. In *Computing, Communication and Security (ICCCS), 2015 International Conference on*, pages 1–4. IEEE, 2015.

[43] Vivian F López, Rubén E Salamanca, María N Moreno, Ana B Gil, and Juan M Corchado. A knowledge-based recommender agent to choosing a competition system. In *Trends in Practical Applications of Agents, Multi-Agent Systems and Sustainability*, pages 143–150. Springer, 2015.

[44] Jie Lu, Dianshuang Wu, Mingsong Mao, Wei Wang, and Guangquan Zhang. Recommender system application developments: a survey. *Decision Support Systems*, 74:12–32, 2015.

[45] Linyuan Lü, Matúš Medo, Chi Ho Yeung, Yi-Cheng Zhang, Zi-Ke Zhang, and Tao Zhou. Recommender systems. *Physics Reports*, 519(1):1–49, 2012.

[46] Michele Manca, Ludovico Boratto, and Salvatore Carta. Design and architecture of a friend recommender system in the social bookmarking domain. In *Science and Information Conference (SAI), 2014*, pages 838–842. IEEE, 2014.

[47] Stuart E Middleton, David De Roure, and Nigel R Shadbolt. Ontology-based recommender systems. In *Handbook on ontologies*, pages 779–796. Springer, 2009.

[48] Tien T Nguyen, Daniel Kluver, Ting-Yu Wang, Pik-Mai Hui, Michael D Ekstrand, Martijn C Willemsen, and John Riedl. Rating support interfaces to improve user experience and recommender accuracy. In *Proceedings of the 7th ACM conference on Recommender systems*, pages 149–156. ACM, 2013.

[49] Sean Owen, Robin Anil, Ted Dunning, and Ellen Friedman. *Mahout in action*. Manning Shelter Island, 2011.

[50] Michael J Pazzani. A framework for collaborative, content-based and demographic filtering. *Artificial Intelligence Review*, 13(5-6):393–408, 1999.

[51] Carlos Porcel, A Tejeda-Lorente, MA Martínez, and Enrique Herrera-Viedma. A hybrid recommender system for the selective dissemination of research resources in a technology transfer office. *Information Sciences*, 184(1):1–19, 2012.

[52] Abinash Pujahari and Vineet Padmanabhan. An approach to content based recommender systems using decision list based classification with k-dnf rule set. In *Information Technology (ICIT), 2014 International Conference on*, pages 260–263. IEEE, 2014.

[53] Stefan Reiterer, Martin Stettinger, Michael Jeran, Wolfgang Eixelsberger, and Manfred Wundara. Advantages of extending wiki pages with knowledge-based recommendations. In *Proceedings of the 15th International Conference on Knowledge Technologies and Data-driven Business*, page 46. ACM, 2015.

[54] Adem Sabic and Mohamed El-Zayat. Building e-university recommendation system. In *Information Management and Engineering (ICIME), 2010 The 2nd IEEE International Conference on*, pages 439–442. IEEE, 2010.

[55] Oren Sar Shalom, Shlomo Berkovsky, Royi Ronen, Elad Ziklik, and Amir Amihood. Data quality matters in recommender systems. In *Proceedings of the 9th ACM Conference on Recommender Systems*, pages 257–260. ACM, 2015.

[56] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th international conference on World Wide Web*, pages 285–295. ACM, 2001.

[57] Sebastian Schelter, Christoph Boden, Martin Schenck, Alexander Alexandrov, and Volker Markl. Distributed matrix factorization with mapreduce using a series of broadcast-joins. In *Proceedings of the 7th ACM conference on Recommender systems*, pages 281–284. ACM, 2013.

[58] Qiang Song, Jian Cheng, and Hanqing Lu. Incremental matrix factorization via feature space re-learning for recommender system. In *Proceedings of the 9th ACM Conference on Recommender Systems*, pages 277–280. ACM, 2015.

[59] Cassandra Targett. Solr wiki: Solrj, August 2013 (accessed November 25, 2016).

[60] Saúl Vargas and Pablo Castells. Rank and relevance in novelty and diversity metrics for recommender systems. In *Proceedings of the fifth ACM conference on Recommender systems*, pages 109–116. ACM, 2011.

[61] Sebastiano Vigna. fastutil: Fast & compact type-specific collections for java, (accessed November 25, 2016).

[62] Jialei Wang, Steven CH Hoi, Peilin Zhao, and Zhi-Yong Liu. Online multi-task collaborative filtering for on-the-fly recommender systems. In *Proceedings of the 7th ACM Conference on Recommender Systems*, pages 237–244. ACM, 2013.

[63] Jun Wang, Arjen P De Vries, and Marcel JT Reinders. Unifying user-based and item-based collaborative filtering approaches by similarity fusion. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 501–508. ACM, 2006.

[64] Jonathan Wintrode, Gregory Sell, Aren Jansen, Michelle Fox, Daniel Garcia-Romero, and Alan McCree. Content-based recommender systems for spoken documents. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 5201–5205. IEEE, 2015.

[65] Wen Wu, Liang He, and Jing Yang. Evaluating recommender systems. In*Digital Information Management (ICDIM), 2012 Seventh International Conference on*, pages 56–61. IEEE, 2012.

[66] Shuang-Hong Yang, Bo Long, Alex Smola, Narayanan Sadagopan, Zhaohui Zheng, and Hongyuan Zha. Like like alike: joint friendship and interest propagation in social networks. In *Proceedings of the 20th international conference on World wide web*, pages 537–546. ACM, 2011.

[67] Zied Zaier, Robert Godin, and Luc Faucher. Evaluating recommender systems. In *Automated solutions for Cross Media Content and Multi-channel Distribution, 2008. AXMEDIS'08. International Conference on*, pages 211–217. IEEE, 2008.

[68] Zui Zhang, Hua Lin, Kun Liu, Dianshuang Wu, Guangquan Zhang, and Jie Lu. A hybrid fuzzy-based personalized recommender system for telecom products/services. *Information Sciences*, 235:117–129, 2013.

[69] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-scale parallel collaborative filtering for the netflix prize. In *Algorithmic Aspects in Information and Management*, pages 337–348. Springer, 2008.

[70] Yong Zhuang, Wei-Sheng Chin, Yu-Chin Juan, and Chih-Jen Lin. A fast parallel sgd for matrix factorization in shared memory systems. In *Proceedings of the 7th ACM conference on Recommender systems*, pages 249–256. ACM, 2013.

[71] Cai-Nicolas Ziegler, Sean M McNee, Joseph A Konstan, and Georg Lausen. Improving recommendation lists through topic diversification. In *Proceedings of the 14th international conference on World Wide Web*, pages 22–32. ACM, 2005.

Appendix A

Solr Ingestion Script

```
#region consts for geo mappings
def get_region_location(key, region):
  if key == 'de':
    key = region
  if key == 'at':
    return '"geo":"48.2,16.35",'
  elif key == 'ch':
    return '"geo":"46.95,7.45",'
  elif key == '1':
    return '"geo":"48.537778,9.041111",'
  elif key == '2':
    return '"geo":"48.7775,11.431111",'
  elif key == '3':
    return '"geo":"52.516667,13.383333",'
  elif key == '4':
    return '"geo":"52.361944,13.008056",'
  elif key == '5':
    return '"geo":"53.083333,8.8",'
  elif key == '6':
    return '"geo":"53.565278,10.001389",'
  elif key == '7':
    return '"geo":"50.666111,8.591111",'
  elif key == '8':
    return '"geo":"53.616667,12.7",'
  elif key == '9':
    return '"geo":"52.756111,9.393056",'
  elif key == '10':
    return '"geo":"51.466667,7.55",'
  elif key == '11':
    return '"geo":"49.913056,7.45",'
  elif key == '12':
    return '"geo":"49.383056,6.833056",'
  elif key == '13':
    return '"geo":"49.383056,6.833056",'
  elif key == '14':
    return '"geo":"51.971111,11.47",'
  elif key == '15':
    return '"geo":"54.47,9.513889",'
  elif key == '16':
    return '"geo":"50.861111,11.051944",'
  else:
    return ''

def addTags(userid):
  try:
    return user_tags[int(userid)]
  except ValueError:
    return ''
  except KeyError:
    return ''

def addToSolr(toAdd):
  toAdd = '[' + toAdd[:-1] + ']'
  req1 = urllib2.Request(url1)
  req1.add_header('Content-type', 'application/json')
  req1.add_data(toAdd)
  resp1 = urllib2.urlopen(req1)
  print resp1.read()
  req2 = urllib2.Request(url2)
  resp2 = urllib2.urlopen(req2)
  print resp2.read()

#get user mappings and initialize dictionaries
print "Creating user mappings..."
i = 0
for i in range(0, 15000000):
  user_tags[int(i)] = ""
  user_click_similarity[int(i)] = ""
  user_click_coocurrence[int(i)] = ""
  user_mapping[int(i)] = ""
i = 0
for row in users_csv_data:
  user_mapping[row[0]] = i
  i = i + 1

#create item tag dictionary
print "Creating␣item␣tag␣dictionary..."
for row in items_csv_data:
  tags = str(row[10]).replace(",", "␣")
  item_tags[row[0]] = tags

#determine tags of what user clicked on
i = 0
print "determing␣user␣click␣tags..."
for row in interactions_csv_data:
  if i is not 0:
    if (str(row[2]) != "4"):
      try:
        user_tags[int(row[0])] += item_tags[row[1]]
      except KeyError:
        continue
  i = i + 1



#determine click similarity-matrix
print "getting␣click␣similarity␣matrix..."
for click_file in click_similarity_files:
  with open(click_file) as file:
    content = file.readlines()
    for c in content:
      tokens = re.split(r'\t|␣', c)
      if len(tokens) is 1:
        continue
      else:
        for t in tokens:
          influence = t.split(":")
          if len(influence) is 2:
            try:
                user_click_similarity[int(tokens[0])] += influence[0] + ' '
            except KeyError:
              continue

#determine click similarity-matrix
print "getting click coocurrence matrix..."
for click_file in click_view_coocurrence_files:
  with open(click_file) as file:
    content = file.readlines()
    for c in content:
      tokens = re.split(r'\t| ', c)
      if len(tokens) is 1:
        continue
      else:
        for t in tokens:
          influence = t.split(":")
          if len(influence) is 2:
            try:
              user_click_coocurrence[int(tokens[0])] += influence[0] + ' '
            except KeyError:
              continue

#
# #fill user table
i = 0
j = 0
user_doc = ''
for row in users_csv_data2:
  if (row[0]) != 'id':
    user_doc += '{' +\
      '"id":"users_'+ str(row[0]) + '",' + '"collection":"users",' +\
      '"orig_user_id":' + str(row[0]) + ',' +\
      '"user_click_coocurrence":"' + user_click_coocurrence[int(row[0])] + '",' +\
      '"user_click_similarity":"' + user_click_similarity[int(row[0])] + '",' +\
      get_region_location(str(row[5]), str(row[6])) +\
      '"tags":"' + addTags(row[0]) + '",' +\
      '"user_id":' + str(i) + '' +\
      '},'
    if (i != 0 and i % 250000 == 0):
      addToSolr(user_doc)
      user_doc = ''
    i = i + 1
addToSolr(user_doc)

#
# #fill items table
i = 0
item_doc = ''
for row in items_csv_data2:
  if i is not 0:
    item_doc += '{' +\
      '"id":"items_'+ str(i) +'",' +\
      '"collection":"items",' +\
      '"item_id":' + row[0] + ',' +\
      get_region_location(str(row[5]), str(row[6])) +\
      '"tags":"' + str(row[10]) + '",' +\
      '"active_during_test":' + str(row[12]) +\
    '},'
    if (i != 0 and i % 250000 == 0):
      addToSolr(item_doc)
      item_doc = ''
  i = i + 1
addToSolr(item_doc)

# #fill interactions table
doc = ''
i = 0
for row in interactions_csv_data:
  if i is not 0:
    doc += '{' +\
      '"id":"interactions_'+ str(i) +'",' +\
      '"collection":"interactions",' +\
      '"user_id":' + str(row[0] if ('' != row[0] and 'NULL' != row[0])
        else -1) + ',' +\
      '"item_id":' + str(row[1] if ('' != row[1] and 'NULL' != row[1])
        else -1) + ',' +\
      '"interaction_type":' + str(row[2] if ('' != row[2] and 'NULL' != row[2])
        else -1) + ',' +\
      '"created_at":' + str(row[3] if ('' != row[3] and 'NULL' != row[3])
        else -1) + '' +\
    '},'
    if (i % 250000 == 0):
            addToSolr(doc)
            doc = ''
    i = i + 1
addToSolr(doc)
```

Appendix B

Java Code

B.1 App.java

```
public static void main( String[] args ) throws SolrServerException,
  IOException, TasteException, InterruptedException {
  app = new App();
  App.users = new Long2ObjectOpenHashMap<>(50000);
  App.items = new Long2ObjectOpenHashMap<>(50000);
  App.solr = new HttpSolrClient(urlString);
  App.prefArray = new FastByIDMap<>(3);
  App.activeItems = new Long2BooleanOpenHashMap(1358098);
  App.numNeighbors = Integer.parseInt(args[1]);
  App.evaluateOnly = Integer.parseInt(args[2]) == 1 ? true : false;
  App.NUM_THREADS = Integer.parseInt(args[3]);
  App.userToOrigMapping = new ArrayList<>();
  App.solrUtils = new SolrUtils();
  App.fileUtils = new FileUtils();

  System.out.println("Starting...");

  if (args[0].equals("1")) {
    app.recommenderScheme1();
  } else if (args[0].equals("2")) {
    app.recommenderScheme2();
  } else if (args[0].equals("4")) {
    app.recommenderScheme4();
  }else if (args[0].equals("5")) {
    app.recommenderScheme5();
  } else {
    System.out.println("Invalid argument to program.\n");
    System.out.println("Arguments:\n");
    System.out.println("<recommendationMode> <numNeighbors>");
    System.exit(1);
  }
}

public void recommenderScheme1() throws SolrServerException, IOException,
TasteException, InterruptedException {
  solrUtils.initializeActivePostingsMap(App.solr, App.activeItems);
  processData();
  solrUtils.getUsers(App.solr, App.users, App.userToOrigMapping);

  String collectionName = null;
  if (App.numNeighbors == 2) {
    collectionName = "tc2";
  } else if (App.numNeighbors == 4) {
    collectionName = "tc4";
  } else if (App.numNeighbors == 8) {
    collectionName = "tc8";
  } else if (App.numNeighbors == 16) {
    collectionName = "tc16";
  } else if (App.numNeighbors == 32) {
    collectionName = "tc32";
  } else if (App.numNeighbors == 64) {
    collectionName = "tc64";
  } else if (App.numNeighbors == 128) {
    collectionName = "tc128";
  } else if (App.numNeighbors == 256) {
    collectionName = "tc256";
  } else if (App.numNeighbors == 512) {
    collectionName = "tc512";
  } else if (App.numNeighbors == 1024) {
    collectionName = "tc1024";
  }

  LLHUserCollaborativeFiltering(false, App.numNeighbors,
      collectionName);
```

```
}

public void recommenderScheme2() throws SolrServerException, IOException,
TasteException, InterruptedException {

  solrUtils.initializeActivePostingsMap(App.solr, App.activeItems);
  processData();
  solrUtils.getUsers(App.solr, App.users, App.userToOrigMapping);

  String collectionName = null;
  if (App.numNeighbors == 2) {
    collectionName = "llh2";
  } else if (App.numNeighbors == 4) {
    collectionName = "llh4";
  } else if (App.numNeighbors == 8) {
    collectionName = "llh8";
  } else if (App.numNeighbors == 16) {
    collectionName = "llh16";
  } else if (App.numNeighbors == 32) {
    collectionName = "llh32";
  } else if (App.numNeighbors == 64) {
    collectionName = "llh64";
  } else if (App.numNeighbors == 128) {
    collectionName = "llh128";
  } else if (App.numNeighbors == 256) {
    collectionName = "llh256";
  } else if (App.numNeighbors == 512) {
    collectionName = "llh512";
  } else if (App.numNeighbors == 1024) {
    collectionName = "llh1024";
  }

  LLHUserCollaborativeFiltering(true, App.numNeighbors, collectionName);
}

public void recommenderScheme4() throws SolrServerException, IOException,
TasteException {
  System.out.println("Running IR stats evaluator for Item-item CF "
      + "using LogLikelihood");

  solrUtils.initializeActivePostingsMap(App.solr, App.activeItems);
```

```
    processData();
    LLHItemCollaborativeFiltering(true, "recommendations4");
    processData();
}

public void recommenderScheme5() throws SolrServerException, IOException,
TasteException {
    System.out.println("Running IR stats evaluator for Item-item CF "
        + "using TanimotoCoefficients");

    solrUtils.initializeActivePostingsMap(App.solr, App.activeItems);
    processData();
    LLHItemCollaborativeFiltering(false, "recommendations5");
    processData();
}

private void processData() throws SolrServerException, IOException,
TasteException {
    System.out.println("Making solr query for interactions...");
    SolrQuery sq = new SolrQuery();
    sq.set("q", "collection:interactions");
    sq.set("fl", "user_id, item_id, interaction_type");
    sq.set("rows", 2000000);
    QueryResponse response = App.solr.query(sq);
    SolrDocumentList solrDocumentList = response.getResults();

    System.out.println("Starting processing of interaction info...");

    HashMap<Integer, ArrayList<Integer>> userToItems =
        new HashMap<Integer, ArrayList<Integer>>();

    // first parse data
    for (SolrDocument d : solrDocumentList) {
      if (Integer.parseInt(
          (d.getFieldValue("interaction_type").toString())) == 4) {
        continue;
      }
      if (userToItems.get(
          Integer.parseInt(d.getFieldValue("user_id")
              .toString())) != null) {
        userToItems.get(
            Integer.parseInt(
                d.getFieldValue("user_id").toString()))
        .add(Integer.parseInt(d.getFieldValue("item_id").toString()));
    } else {
        ArrayList<Integer> itemList = new ArrayList<Integer>();
        itemList.add(
            Integer.parseInt(
                d.getFieldValue("item_id").toString()));
        userToItems.put(
            Integer.parseInt(
                d.getFieldValue("user_id").toString()),
            itemList);
    }
}

System.out.println("...done!");

System.out.println("Putting info into preference array...");

// put in format for preference array
for (int i = 0; i < 150000; i++) {
    PreferenceArray p = null;
    if (userToItems.containsKey(i)) {
        for (int j = 0; j < userToItems.get(i).size(); j++) {
            if (j == 0) {
                p = new GenericUserPreferenceArray(
                    userToItems.get(i).size());
                p.setUserID(0, i);
            }
            p.setItemID(j, userToItems.get(i).get(j));
        }
        prefArray.put(i, p);
    } else {
        p = new GenericUserPreferenceArray(0);
        p.setUserID(0, i);
        prefArray.put(i, p);
    }
}

System.out.println("...Done!");
```

```
  System.out.println("Creating data model.");
  App.dataModel = new GenericDataModel(prefArray);
}

private void LLHUserCollaborativeFiltering(boolean useLLH,
    int neighborhoodSize, String collectionName) throws TasteException,
SolrServerException, IOException {
  if (useLLH) {
    App.userSimilarity = new LogLikelihoodSimilarity(App.dataModel);
  } else {
    App.userSimilarity =
        new TanimotoCoefficientSimilarity(App.dataModel);
  }
  App.neighborhood = new NearestNUserNeighborhood(neighborhoodSize,
      App.userSimilarity, App.dataModel);
  App.recommender = new BasicFilteringRecommender(App.dataModel,
      App.neighborhood, App.userSimilarity);

  ArrayList<Long> targetUsers = fileUtils.getTargetUsers();

  ExecutorService executor =
      Executors.newFixedThreadPool(NUM_THREADS);
  int workLoad = targetUsers.size() / NUM_THREADS;
  for (int i = 0; i < NUM_THREADS; i++) {
    ArrayList<Long> users = new ArrayList<>();
    for (int k = i * workLoad; k < (i + 1) * workLoad; k++) {
      users.add(targetUsers.get(k));
    }
    executor.execute(new ParallelRecommendations(users,
        collectionName));

    executor.shutdown();
    while (!executor.isTerminated()) {}

    fileUtils.writeRecommendationSetToCSV(collectionName,
      App.solr, App.userToOrigMapping);
    System.exit(0);
  }
}

private void LLHItemCollaborativeFiltering(boolean useLLH,
  String collectionName) throws TasteException, SolrServerException,
IOException {
  if (useLLH) {
    App.itemSimilarity = new LogLikelihoodSimilarity(App.dataModel);
  } else {
    App.itemSimilarity = new TanimotoCoefficientSimilarity(App.dataModel);
  }
  App.recommender = new
    GenericBooleanPrefItemBasedRecommender(App.dataModel, App.itemSimilarity);

  ArrayList<Long> targetUsers = fileUtils.getTargetUsers();

  ExecutorService executor = Executors.newFixedThreadPool(NUM_THREADS);
  int workLoad = targetUsers.size() / App.NUM_THREADS;
  for (int i = 0; i < NUM_THREADS; i++) {
    ArrayList<Long> users = new ArrayList<>();
    for (int k = i * workLoad; k < (i + 1) * workLoad; k++) {
      users.add(targetUsers.get(k));
    }
    executor.execute(new ParallelRecommendations(users, collectionName));
  }

  executor.shutdown();
  while (!executor.isTerminated()) {}

  fileUtils.writeRecommendationSetToCSV(collectionName,
    App.solr, App.userToOrigMapping);
  System.exit(0);
}
```

B.2 BasicFilteringRecommender.java

```
private Recommender delegate;

public BasicFilteringRecommender(DataModel model, UserNeighborhood neighborhood,
  UserSimilarity similarity) {
  delegate = new GenericBooleanPrefUserBasedRecommender(model, neighborhood,
  similarity);
}
```

```
public void refresh(Collection<Refreshable> alreadyRefreshed) {
delegate.refresh(alreadyRefreshed);
}

public List<RecommendedItem> recommend(long userID, int howMany)
  throws TasteException {
  IDRescorer rescorer = new BasicRescorer(userID);
  return recommend(userID, howMany, rescorer);
}

public List<RecommendedItem> recommend(long userID, int howMany,
  IDRescorer rescorer) throws TasteException {
  return delegate.recommend(userID, howMany, rescorer);
}

public float estimatePreference(long userID, long itemID)
  throws TasteException {
  return delegate.estimatePreference(userID, itemID);
}

public void setPreference(long userID, long itemID, float value)
  throws TasteException {
  delegate.setPreference(userID, itemID, value);
}

public void removePreference(long userID, long itemID)
  throws TasteException {
  delegate.removePreference(userID, itemID);
}

public DataModel getDataModel() {
  return delegate.getDataModel();
}

public List<RecommendedItem> recommend(long userID, int howMany,
  boolean includeKnownItems) throws TasteException {
  IDRescorer custom = new BasicRescorer(userID);
  return delegate.recommend(userID, howMany, custom, includeKnownItems);
}

public List<RecommendedItem> recommend(long userID, int howMany,
  IDRescorer rescorer, boolean includeKnownItems)
  throws TasteException {
  return delegate.recommend(userID, howMany, rescorer, includeKnownItems);
}
```

B.3 BasicRescorer.java

```
private long userID;

public BasicRescorer(long userID) {
  this.userID = userID;
}

public double rescore(long id, double originalScore) {
  if (!isFiltered(id)) {
    return originalScore;
  } else {
    return Double.NaN;
  }
}

public boolean isFiltered(long id) {
  if (App.activeItems.containsKey(id)) {
    if(App.activeItems.get(id)) {
      SolrQuery sq = new SolrQuery();
      sq.set("q", "collection:interactions");
      sq.addFilterQuery("user_id:" + this.userID);
      sq.addFilterQuery("item_id:" + id);
      sq.addFilterQuery("interaction_type:4");
      sq.set("rows", 1);
      try {
        QueryResponse response = App.solr.query(sq);
        if (response.getResults().size() != 0) {
          return true;
        }
      } catch (SolrServerException e) {
        return false;
      } catch (IOException e) {
        return false;
      }
      return false;
    }
    return true;
  } else {
    return false;
  }
}
```

B.4 MultimodalRecommender.java

```
public static void main(String[] args)
  throws IOException, InterruptedException, SolrServerException {
  MultimodalRecommender q = new MultimodalRecommender();
  q.solr = new HttpSolrClient(q.urlString);
  q.users = new Long2ObjectOpenHashMap<>();
  q.userToOrigMapping = new ArrayList<Long>();
  q.results = new HashMap<>(15000000);
  solrUtils = new SolrUtils();
  fileUtils = new FileUtils();
  solrUtils.getUsers(solr, users, userToOrigMapping);

  q.MultimodalRecommender();
}

private void MultimodalRecommender() throws IOException {
  String userClickHistory = "";
  String userBookmarkAndReplyHistory = "";
  String userClickTags = "";
  String geoLoc = "";
  ArrayList<Long> targetUsers = new ArrayList<>();
  targetUsers = fileUtils.getTargetUsers();

  for (Long userId : targetUsers) {
    SolrQuery sq = new SolrQuery();
    SolrQuery sq2 = new SolrQuery();

    sq.set("q", "collection:interactions");
    sq.set("fq", "user_id:" + userId);
    try {
      QueryResponse response = solr.query(sq);
```

```
for (SolrDocument d2 : response.getResults()) {
  if (d2.getFieldValue("interaction_type").toString().equals("1")) {
    userClickHistory += d2.getFieldValue("item_id").toString() + "␣";
  } else if (d2.getFieldValue("interaction_type").toString().equals("2") ||
      d2.getFieldValue("interaction_type").toString().equals("3")) {
    userBookmarkAndReplyHistory +=
      d2.getFieldValue("item_id").toString() + "␣";
  }
}

sq2.set("q", "collection:users");
sq2.set("fq", "orig_user_id:" + userId);
sq2.set("fl", "tags,␣geo");
QueryResponse response2 = solr.query(sq2);
try {
  userClickTags = response2.getResults().get(0).getFieldValue("tags").toString();
} catch (NullPointerException e) {
  e.printStackTrace();
} catch (IndexOutOfBoundsException e) {
  e.printStackTrace();
}
try {
  geoLoc = response2.getResults().get(0).getFieldValue("geo").toString();
} catch (NullPointerException e) {
  e.printStackTrace();
} catch (IndexOutOfBoundsException e) {
  e.printStackTrace();
}

SolrQuery sq3 = new SolrQuery();

String MultimodalRecommender1 =
  userClickHistory.replace("[", "").replace("]", "");
String MultimodalRecommender2 =
  userBookmarkAndReplyHistory.replace("[", "").replace("]", "");
String MultimodalRecommender3 =
  userClickTags.replace("[", "").replace("]", "");

String finalMultimodalRecommender = "collection:users␣AND␣(";
if (!MultimodalRecommender1.equals("")) {
  finalMultimodalRecommender +=
        "user_click_similarity:" + MultimodalRecommender1 + " OR ";
}
if (!MultimodalRecommender2.equals("")) {
    finalMultimodalRecommender +=
        "user_click_coocurrence:" + MultimodalRecommender2 + " OR ";
}
if (!MultimodalRecommender3.equals("")) {
    finalMultimodalRecommender += "tags:" + MultimodalRecommender3;
}
finalMultimodalRecommender += ")";

sq3.set("q", finalMultimodalRecommender);
sq3.setSort("score", SolrQuery.ORDER.desc);

QueryResponse response3 = solr.query(sq3);

Map<String, Integer> resultHist = new HashMap<>();
for (SolrDocument d3 : response3.getResults()) {
    try {
        String[] clickSimVals =
            d3.getFieldValue("user_click_similarity").toString().split(" ");
        for (String value : clickSimVals) {
            if (resultHist.containsKey(value)) {
                resultHist.put(value, resultHist.get(value) + 1);
            } else {
                resultHist.put(value, 1);
            }
        }
    } catch (NullPointerException e) {
        e.printStackTrace();
    }
    try {
        String[] clickCoocVals =
            d3.getFieldValue("user_click_coocurrence").toString().split(" ");
        for (String value : clickCoocVals) {
            if (resultHist.containsKey(value)) {
                resultHist.put(value, resultHist.get(value) + 1);
            } else {
                resultHist.put(value, 1);
            }
        }
```

```
      } catch (NullPointerException e) {
      }
    }
    Map<String, Integer> sortedMap =
        resultHist.entrySet().stream()
            .sorted(Map.Entry.comparingByValue())
            .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue,
                (e1, e2) -> e1, LinkedHashMap::new));
    SolrInputDocument document = new SolrInputDocument();
    document.addField("id", "multimodal_" + userId);
    document.addField("user_id", userId);
    document.addField("collection", "multimodal");
    Object[] top = sortedMap.keySet().toArray();
    int counter = 0;
    for (int i = top.length - 1; i > 0; i--) {
      if (CheckValidItem(top[i].toString())) {
        document.addField("items", top[i].toString());
        counter++;
        if (counter == 30) {
          break; // we have all recs
        }
      }
    }
    solr.add(document, 25000);
    solrUtils.commit(solr);
  } catch (SolrServerException e) {
    e.printStackTrace();
  } catch (IOException e) {
    e.printStackTrace();
  } catch (HttpSolrClient.RemoteSolrException e) {
    // no recs, use back up method with geo
    SolrQuery sq3 = new SolrQuery();
    sq3.set("q", "collection:items");
    sq3.addFilterQuery("{!geofilt}");
    sq3.add("sfield", "geo");
    sq3.add("pt", geoLoc);
    sq3.add("d", "100");
    sq3.addFilterQuery("active_during_test:1"); // only valid recs
    sq3.setSort("geodist()", SolrQuery.ORDER.asc);

    try {
            QueryResponse resp = solr.query(sq3);
            int i = 0;
            SolrInputDocument document = new SolrInputDocument();
            document.addField("id", "multimodal_" + userId);
            document.addField("user_id", userId);
            document.addField("collection", "multimodal");
            for (SolrDocument d : resp.getResults()) {
              if (++i == 30) {
                break;
              }
              document.addField("items", d.getFieldValue("item_id").toString());
              solr.add(document, 25000);
              solrUtils.commit(solr);
            }
          } catch (SolrServerException e1) {
            e1.printStackTrace();
          }
        }
      }

      try {
        fileUtils.writeRecommendationSetToCSV("multimodal", solr, userToOrigMapping);
      } catch (SolrServerException e) {
        e.printStackTrace();
      }
    }

    private boolean CheckValidItem(String itemId) {
      SolrQuery sq = new SolrQuery();
      sq.set("q", "collection:items");
      sq.setFilterQueries("item_id:" + itemId);
      sq.setFilterQueries("active_during_test:1");
      QueryResponse r = null;
      try {
        r = solr.query(sq);
      } catch (SolrServerException e) {
        e.printStackTrace();
      } catch (IOException e) {
        e.printStackTrace();
      }
      if (r.getResults().size() == 0) {
```

```
    return false;
  } else {
    return true;
  }
}
```