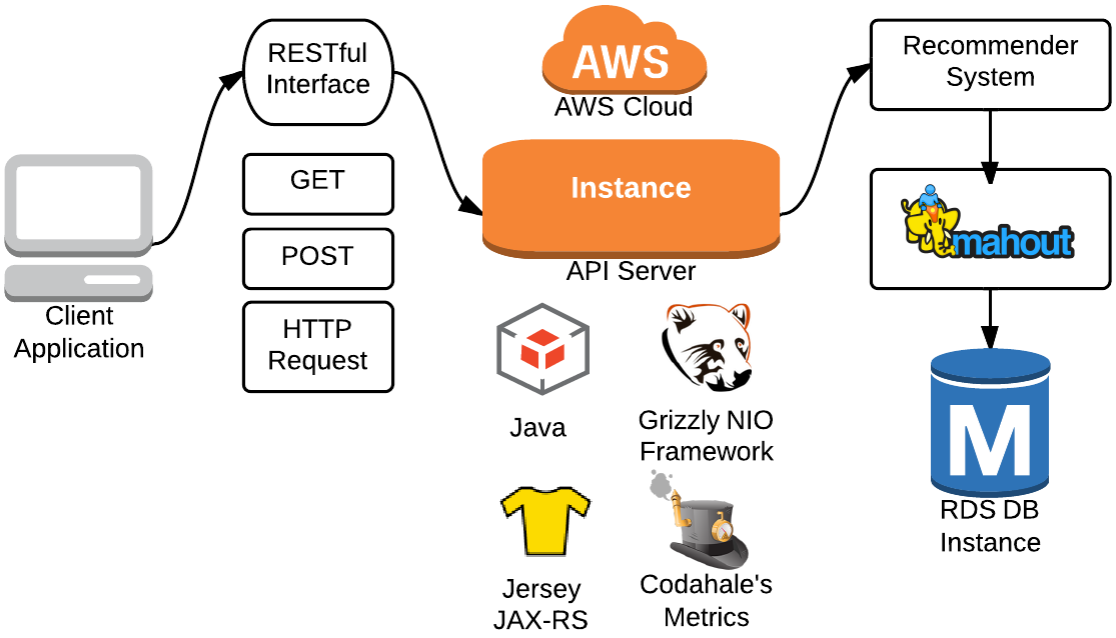
Evaluating and Implementing Recommender Systems As Web Services Using Apache Mahout



By: Peter Casinelli

Advisor: Sergio Alvarez

CONTENTS

Evaluating and Implementing Recommender Systems As Web Services Using Apache Mahout

Boston College Computer Science Senior Thesis

By: Peter Casinelli

Advisor: Sergio Alvarez

A B S T R A C T

In recent years there has been a dramatic increase in the amount of online content. Recommender systems software has emerged to help users navigate through this increased content, often leveraging user-specific data that is collected from users. A recommender system helps a user make decisions by predicting their preferences, during shopping, searching, or simply browsing, based on the user's past preferences as well as the preferences of other users. This thesis explores different recommender system algorithms such as User-User Collaborative and Item-Item Collaborative filtering using the open source library Apache Mahout. We simulate recommendation system environments in order to evaluate the behavior of these collaborative filtering algorithms, with a focus on recommendation quality and time performance. We also consider how recommender systems behave in real world applications. We explore the implementation of a web service that serves as a front end to a recommender system, keeping in mind our evaluation results, as well as ease of access to applications, and the overall user experience.

1 AN INTRODUCTION TO RECOMMENDER SYSTEMS

1.1 What Are Recommender Systems?

Throughout the internet there are web sites, applications, and systems being built with a focus on user interaction and data. The users of these systems expect to be introduced to new content, to be recommended content that their friends like, and want interfaces through which they can submit feedback to improve these recommendations. Recommender systems (RSs) are the tools and techniques that address these demands by utilizing user data and algorithms to suggest new items that will be of use to users [1, 2]. A RS can provide suggestions for products to buy, books to read, places to eat, or movies to watch.

1.2 Recommender System Components

Recommender systems are often comprised of several components known as users, items, preferences/ratings, and neighborhoods. **Items** are the things or objects that are being recommended to a user. For example, items are often products, news articles, songs or movies. These items can be characterized by their respective metadata that include relevant titles, tags, or keywords. For example, news articles can be characterized by content category, songs can be characterized by artists and genre, and movies can be characterized by genre and director. **Users** are the people who are being recommended items. They often need assistance or guidance in choosing an item within an application and use recommendation to help them make an informed and hypothetically better decision. A user model can be built over time in an effort to make better recommendations for each particular user. This user model acts as a profile in which preferences and actions are encoded and is representative of the history of a user and their interactions with items within the RS. These interactions are known as **preferences**. **Preferences** can be interpreted as the user's opinion of an item in a RS and can be both explicit or implicit. Preferences are often categorized as ratings if a RS provides an interface to rate items. A **rating** is a type of *explicit* preference that

represents a relationship between a user and an item. Every rating can describe, for example, how a user feels about certain items. An example of an explicit rating is a user rating a movie with five stars. From this rating, the RS can definitively conclude that the user likes the movie item. An implicit preference can be a user clicking on a link or skipping a video. In these examples, we can infer data from these implicit preferences and assume that the user may like an item if they click on its link, or do not like a video that they skip. A **neighborhood** relates users and their preferences and represents a group of similar users. In collaborative filtering (CF) environments, which will be discussed later, neighborhoods of similar users help a RS decide on items to recommend to a user based on users with similar tastes [2].

1.3 Recommender System Dimensions

Every RS is uniquely characterized by several dimensions that can provide insight into how and why a RS has been implemented. Dimensions such as *domain*, *purpose*, *context*, *personalization level*, *interface,* and *algorithm selection* can explain a recommender system's goals. The domain of recommendation can help identify the components of a RS; for example, an application that recommends movies establishes a movie as the item component of the RS. The purpose of these recommendations are often to encourage movie watching users and to help users discover new movies they may want to watch. The context of such a RS can be a user browsing new movies or movies they have already seen in order to be suggested similar movies they may like. *Personalization* of a RS helps explain the choice of the RS algorithm that is implemented. For example, if the personalization level is based on ratings of movies on a five star scale, a User-User CF algorithm can be used to suggest movies that similar users have rated. Two important dimensions of RSs are the *interfaces* through which preferences are inputted into the RS and how recommendations are outputted from the RS. Lastly, one of the most influential dimensions on a RS is the algorithm used to make a recommendation. Some common algorithms include Non-Personalized, Content-Based, Collaborative, and Hybrid filtering. This thesis primarily examines Non-Personalized, User-User CF, and Item-Item CF filtering algorithms [3].

1.4 Recommendation Algorithms

The definition, components, and dimensionality help us describe and understand RSs. This leads to a discussion of the specific implementation of commonly used RS algorithms that are evaluated and implemented in this thesis. Every RS attempts to predict items that a user will find most relevant and useful. While this concept is common across all types of RSs, the manner by which a RS calculates relevance and usefulness varies.

The amount and type of available data about RS components such as users, items, and preferences often dictate how this relevance and usefulness is calculated and ultimately impacts a RS algorithm selection. When data about a user and their preferences are lacking, a Non-Personalized RS can be an appropriate algorithm selection. A **Non-Personalized** RS algorithm will rely on the overall data about popular items amongst all users and generate recommendations such as a Top-N list of most popular items (*see Figure 1.1*). Non-personalized recommendation algorithms do not provide personalized or diverse recommendations to different users based on past preferences of users. Instead, the RS assumes an item that is liked by most users will also be liked by a generic user [2]. While not heavily researched, non-personalized algorithms provide a simple and effective interface to provide recommendations to users when they lack previous preferences, also known as the cold start problem.

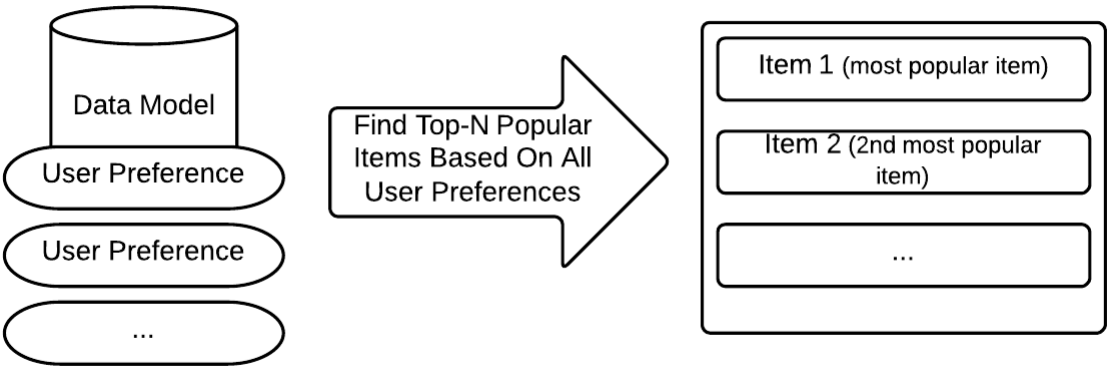
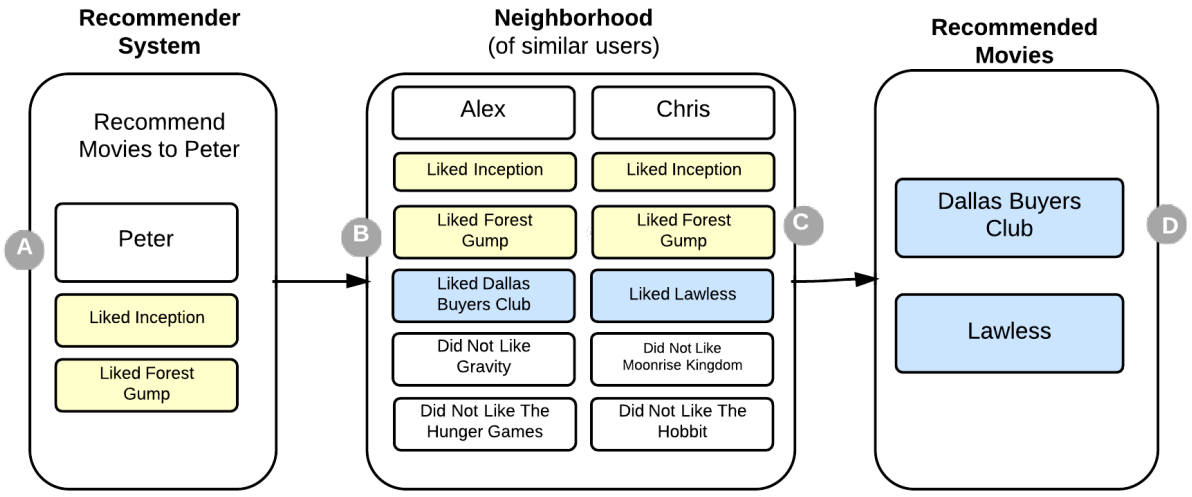


Figure 1.1 A non-personalized algorithm uses all data from a recommender system's data model to produce recommendations such as a Top-N list of the most popular items. This is a generic, non-personalized recommendation since recommendations are given to a user without taking their specific preferences into consideration; only the collective of all user preferences are used.

In recent years, an algorithm known as **Collaborative Filtering** has become commonly implemented in RSs. CF algorithms use the similarity between data such as the preferences of users, neighborhoods, and items in order to more effectively recommend items from a growing set of choices [4]. This thesis examines both **User-User** and **Item-Item Collaborative Filtering** algorithms and the evaluation of their recommendation to users.

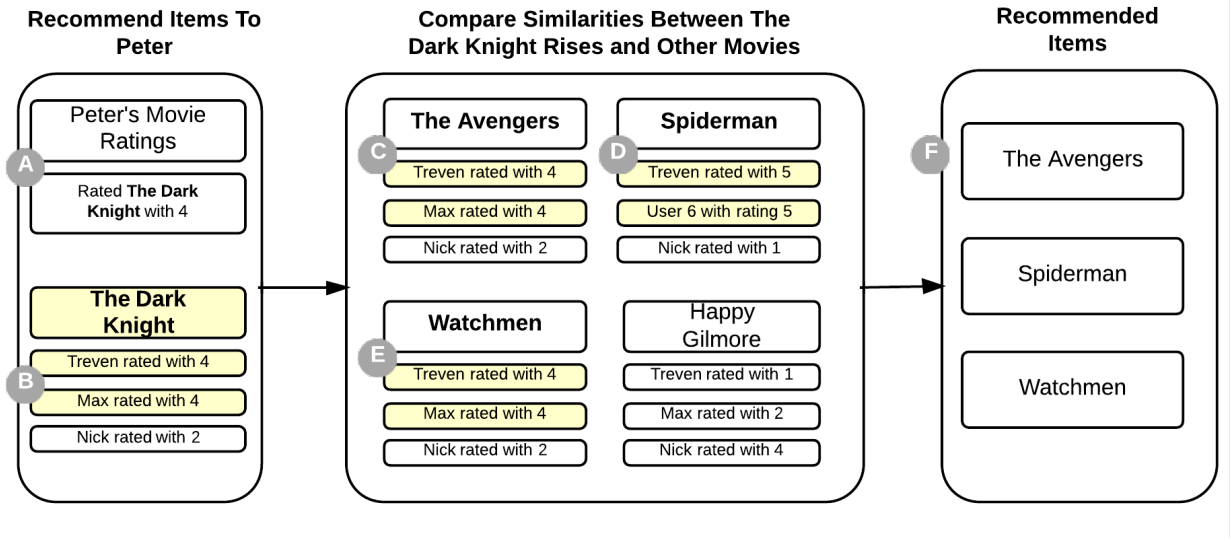
If there are user preference data in a recommender system's model, it is possible to make personalized recommendations based on similarities of user tastes or preferences. In a **User-User CF** RS (*shown in Figure 1.2*), correlations can be identified between different users based on past preferences that are similar in order to make predictions on what each user will like in the future. If two users have rated many items similarly in the past, they may be considered in the same neighborhood. Often, a neighborhood of similar users is built by a RS and used to help recommend items [2]. User-User CF has a personalized advantage over Non-Personalized RSs; the recommendations from User-User CF will be specific to each user and will adapt with the user as they introduce new preferences into the RS.

Figure 1.2 A User-User CF RS recommends items to a user by building a neighborhood of similar users and recommending items based on these neighbors' past ratings. In this figure, the RS is being asked to produce recommended items for a user named Peter who has liked, or positively rated, the movies Inception and Forest Gump as shown by **A**. The RS first builds a neighborhood of similar users: Alex and Chris, who have both expressed similar positive ratings for the same movies as Peter as shown in **B-C**. Since Alex and Chris have also liked the movies Dallas Buyers Club and Lawless, the RS may recommend these items to Peter as shown in **D**. The RS will consider these movies more likely to be positively rated by Peter than other movies since Peter's neighbors have rated them positively. This is based on the intuition that similar users like similar items.



In an **Item-Item CF** RS (*shown in Figure 1.3*), the similarities between items are used in order to make recommendations. Rather than building a neighborhood and making recommendations based on similar users, correlations are made between items' preferences. For example, in order to recommend a new item to user **u**, all of the items for which **u** has a preference are compared to all other items **i** using a similarity algorithm. The intuition is that **u** will be recommended items that are most similar to items **u** has already rated based on past preferences [5]. Item-Item CF can be advantageous because of the smaller scale of items; for example, items tend to grow at a slower pace than users and items also change less over time than users. The implementations of algorithms for calculating User-User, Item-Item CF, user similarity, and item similarity are discussed in section ***2.2 Mahout Recommender Systems*** and ***2.3 Mahout Similarity Algorithms.***

Figure 1.3 A Item-Item CF RS makes recommendations based on similarities between items. In this diagram, **A** shows that Peter has rated the movie The Dark Knight with a 4. If the RS must provide Item-Item CF recommendations to Peter, it will attempt to recommend movies that are similar to the movie The Dark Knight since Peter has positively rated this movie in the past. In this example, we see through **B** that Treven and Max have positively rated The Dark Knight with a 4 while Nick has rated it with a 2. In **C-E**, we see that Treven, Max, and Nick have rated The Avengers, Spiderman, and Watchmen similarly in comparison with The Dark Knight. Therefore, the RS will recommend these similar movies as shown in **F**. This is based on the intuition that users will like items that are similar to items they have liked in the past.



Lastly, a **Hybrid** RS combines algorithms to produce recommendations. Non-personalized and collaborative filtering are not exclusive algorithms; a hybrid RS can compensate for when other RSs do not have enough data to produce quality recommendations. This thesis does not go into hybrid RSs in detail, but they are important in real world applications of RSs where problems such as cold start, data sparsity, and scaling are realities.

1.5 Examples of Recommender Systems

In order to better understand recommender systems, their dimensions, and algorithms, there are several helpful examples of RSs used on websites.

Amazon.com

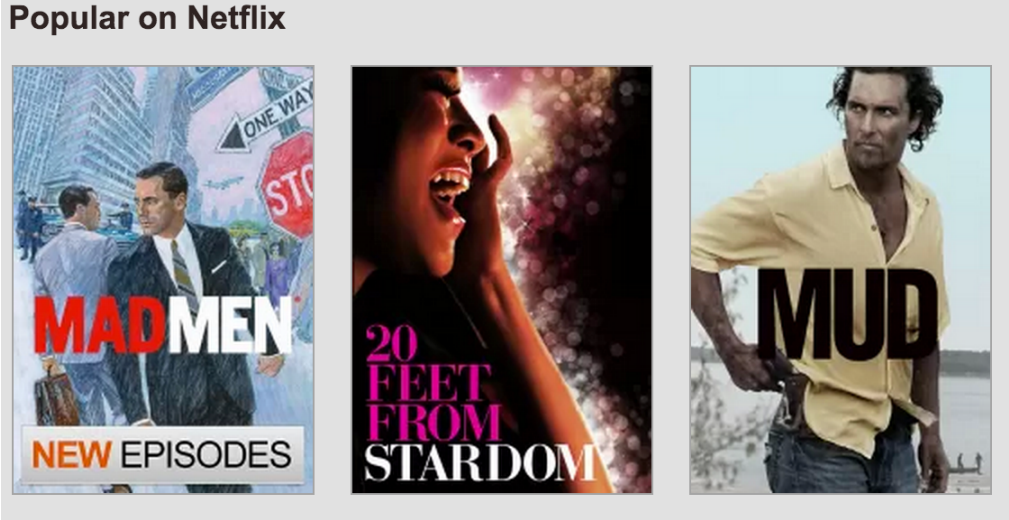
Amazon.com, one of the most popular e-commerce web sites on the internet, has pioneered collaborative filtering recommender systems that consumers now expect when shopping. In *Figure 1.4*, similar books are being recommended to a user browsing for books about recommender systems. Using our framework of RSs, dimensions, and algorithms we can extract information about the RS that is being used. This is an Item-Item CF algorithm recommending items, that are books, to users who are browsing web pages that contain information about books and imply this user is thinking about purchasing books about recommender systems.

Figure 1.4 Amazon often provides recommendations to users by displaying similar items that other users have purchased.

Netflix.com

Netflix.com has built an application that revolves around movie recommendation. After logging into Netflix, a user is immediately recommended movies based on their preferences, top-n lists of movies, and movies that are similar to other movies a user has watched. In *Figure 1.5*, a simple example of a Top-N Non-Personalized algorithm is indicative of a RS that is recommending items, in this case movies, to users who want to watch movies or discover new movies.

Figure 1.5 Netflix movie recommendations that are presented to a user upon login. These are non-personalized Top-N recommendations that represent the most popular movies on Netflix.



1.6 Addressing Common Recommender System Problems

As we have discussed, there are several components and dimensions of RSs that contribute to a complex system that must be designed with factors such as recommendation accuracy, coverage, scale, and speed kept in mind. This thesis explores common problems with designing and implementing RSs through experiments and simulations that evaluate and implement RSs using various algorithms.

In particular, the experiments in the following sections are concerned with the variability of data such as users, items, and preferences, how RSs respond to issues such as a lack of user preferences for new users (the cold start problem), as well as how a RS adapts as the data model grows with new users, items, and preferences.

Evaluation, however, is only part of the process of understanding how RSs behave in different environments. In a production environment, RSs must be able to provide accurate recommendations in a reasonable amount of time in order to maintain consistent user interaction and a positive user experience. For example, a user on Netflix will not wait 10 seconds to receive movie recommendations. In Section 3, we evaluate a RS exposed as a web service and utilize open source libraries to measure recommendation response times to client applications.

2 EVALUATING RECOMMENDER SYSTEMS USING APACHE MAHOUT

2.1 What is Apache Mahout?

Apache Mahout is an open source machine learning library that consists of a framework of tools that allow developers to create powerful and scalable recommender, clustering, and classification applications. Mahout started in 2008 as a spin off technology from the Apache Lucene project, which was primarily concerned with content search and information retrieval technologies. Since there was much overlap between the techniques and algorithms used in the projects such as clustering and classification, Mahout became its own project and also included an open source collaborative filtering project known as Taste. Today, the Mahout library is suitable for applications that require scaling to large datasets because it was opened to contributions for implementations that run on top of Apache Hadoop and now will accept implementations that run on top of Apache Spark *(see Section **4.3 Future Work**)*. This thesis primarily examines Apache Mahout and implementing a recommender system using Mahout's collaborative filtering recommender engine libraries [6].

2.2 Mahout Recommender Systems

The Mahout library contains several commonly used RSs. For the purposes of this thesis, there are specific RSs that are measured, explored, and exposed as web services. This section discusses how Mahout has implemented the following RSs: **Non-Personalized Item Average**, **User-User Collaborative Filtering**, and **Item-Item Collaborative Filtering.**

2.2.1 Non-Personalized Item Average RS

The non-personalized item average recommender estimates a user's preference for an item by calculating the average of all of the known preferences for that item. This RS is non-personalized since none of the user's past preferences are considered to rescore the recommendations [7]. The pseudo code to estimate a user's preference for an item is:

Pseudo Code to Estimate User Preference in Non-Personalized RSs

```
for every preference p for item i
        include p in a running average of all p's for i
return the running average of all p's for I [7]
```

2.2.2 User-User Collaborative Filtering RS

The User-User CF RS first creates a neighborhood n_u of users that are similar to user u based on similarity algorithms that are described below in **Section 3.3**. Then, using n_u, the RS estimates the user u's preference for item i by taking into consideration all of the preferences of neighbors in n_u that have rated item i. User-User CF therefore focuses on similarities between users' preferences [8]. The pseudo code for Mahout's User-User CF to estimate a user **u**'s preference for an item **i** is:

Pseudo Code to Estimate User Preference in User-User CF RSs

for every other user w
    compute a similarity s between user u and user w

    store users with the greatest similarity s in a neighborhood n

for every neighbor w_n in n

    if w_n has a preference for item i

        retrieve this preference value p, apply a weight with value of s, and incorporate it into u's preference for item i

return u's normalized preference for item i [8, 9]

The similarity algorithm used to create this neighborhood is discussed in the next section, **3.3 Mahout Similarity Algorithms**.

2.2.3 Item-Item Collaborative Filtering RS

The Item-Item CF RS will also recommend an item ***i*** to user ***u*** by using a similarity algorithm. It differs from User-User CF because the RS focuses on the similarity between different items' preferences rather than the similarity between different users' preferences. The pseudo code for Mahout's Item-Item CF to estimate a user **u**'s preference for an item **i** is:

Pseudo Code to Estimate User Preference in Item-Item CF RSs

for each item j that user u has a preference for, calculate the similarity s between j's preferences and item i's preferences

for each j that is similar with i
    calculate a weighted preference p_w for i by multiplying u's preference for j by **s**

    incorporate p_w into an overall preference value p_o

return a normalized p_o [10, 11]

When making Item-Item CF based recommendations, Mahout uses this estimation process by applying this pseudo code to every other item in the data model.

2.3 Mahout Similarity Algorithms

In the CF algorithms recently mentioned, there is a commonality between how users and items are determined to be similar to other users and items. The Mahout library has implemented several widely used similarity algorithms and allow developers to plug them into the CF RSs in order identify similar neighborhoods for users or calculate similarities between items. While Mahout has implemented similarity algorithms including Euclidean Distance Similarity, Tanimoto Coefficient Similarity, and Uncentered Cosine Similarity, for the purposes of this thesis the **Pearson Correlation Coefficient Similarity** and the **Log Likelihood Ratio Similarity** algorithms are described and measured using the data sets discussed in ***4.1 Data Sets Used***.

In the following similarity algorithms, user preference values are the basis from which similarities can be calculated between different users and different items. Therefore, both of these similarity algorithms can be used in User-User and Item-Item CF RSs.

2.3.1 Pearson Correlation Coefficient

The Pearson Correlation Coefficient (PCC) determines the similarity between two users or items by measuring the tendency of two series of preferences to move together in a proportional and linear manner [12]. In this thesis' experiments, the PCC similarity algorithm only considers preferences on which both users or items overlap. It attempts to find each users' or items' deviations from their average rating while identifying linear dependencies between two users or items. The formula uses actual preference values, in our case the movie rating value, to find correlation between users or items, and gives larger weights to users or items that agree often especially in extreme cases [3]. The PCC similarity calculation used in this thesis is:

$$PC(w,u) = \frac{\sum_i (r_{w,i} - \bar{r}_w)(r_{u,i} - \bar{r}_u)}{\sqrt{\sum_i (r_{w,i} - \bar{r}_w)^2 \sum_i (r_{u,i} - \bar{r}_u)^2}} \quad [5]$$

Where **w** and **u** represent the two users or items for which the coefficient is being calculated, **i** is an item, $\mathbf{r}_{w,i}$ and $\mathbf{r}_{u,i}$ are individual ratings from **w** and **u** for **i**, and $\bar{r}_w$ and $\bar{r}_u$ are average ratings for user (or item) **w** and **u**, respectively.

Helpful explanations of the issues with the PCC similarity algorithm are discussed in the book *Mahout in Action*. For example, PCC does not take into consideration the number of overlapping preferences. This is intuitively naïve; for example if two users have rated 10 movies similarly, these users will have a lower similarity than two users who rated only two movies very similarly. As a result of these problems, the PCC may not always provide the most accurate recommendation, which is exemplified in later experiments [13]. In the next section, we discuss the Log Likelihood Ratio, which does not consider preference values but does take into consideration statistics such as the number of overlapping preferences.

2.3.2 Log Likelihood Ratio Similarity

The Log Likelihood Ratio (LLR) was created by Ted Dunning in his paper, "Accurate Methods for the Statistics of Surprise and Coincidence." The LLR relies on calculating similarity between two users or items based on statistics that revolve around occurrences related to these users or items. LLR focuses on events where these users or items overlap in preferences, events where both users or items have preferences where the compared user or item does not, and events where both users or items do not have preferences. A helpful explanation and chart is available at [14] and summarizes these events [15, 16].

The LLR predicts how unlikely the overlap between preferences is due to chance or if the overlap represents a genuine similarity. For example, if two users have five preferences in common, but have both only introduced 20 preferences into the data model, they will be considered more similar than two users who have five preferences in common but have both introduced over 100 preferences into the data model [16].

In Mahout, the LLR is used to calculate similarities between items and users. When calculating these similarities, it never considers the actual preference value; LLR only considers the events recently discussed and uses the ratio calculated using the LLR formula as a weight to estimate preferences to users. See **Sections 3.2.2 and 3.3.3** for pseudo code that shows the similarity value being used as a weight in estimating a preference.

2.4 Experiments, Results, and Discussion

2.4.1 Data Sets Used

In the experiments with the Apache Mahout library, the GroupLens' MovieLens data sets from [17] are used. This collection of data sets includes data about movies including users, movies, and movie ratings from users. Some of the data sets include metadata about users and movies, but this thesis does not utilize those features. In some experiments, different data set sizes are used. These include the MovieLens 100k (ML100k) data set that contains around 100,000 ratings from ~1,000 users on ~1,700 movies, the MovieLens 1M (ML1M) data set that contains around 1,000,000 ratings from ~6,000 users on ~4,000 movies, and finally the MovieLens 10M (ML10M) data set that contains around 10,000,000 ratings and ~100,000 tags applied to ~10,000 movies by ~72,000 users [17].

Table 2.1 Data set sizes and their respective number of preferences, users, and items. In the context of the data set, the items are movies and the preferences are users' ratings on movies. The rating can be a whole number from one to five.

Data Set	Preferences	Users	Items
ML100K	100,000	943	1,682
ML1M	1,000,209	6,040	3,383
ML10M	10,000,054	71,567	10,681

2.4.2 Evaluation and Metrics

When evaluating a RS it is necessary to use subsets of a data set in order to estimate and verify recommendation. In the evaluation process, **training data** refers to the subset of data that is used to "build" a RS; with this training data, the RS evaluator will attempt to estimate a user's preference for an item. After the RS estimates this preference, it uses actual user preference data from the evaluation data set in order to determine how accurate the estimated preference was. **Evaluation data** is therefore the subset on which deviations from actual and predicted user rates are measured.

There are several metrics by which a RS can be evaluated and interpreted for accuracy. In the following experiments, this thesis evaluates the different datasets using three common evaluation metrics: Mean Absolute Error (MAE), Root Mean Square Error (RMSE), and Coverage.

MAE and RMSE are known as predictive accuracy or statistical accuracy metrics because they represent how accurately a RS estimates a user's preference for an item. In our movie dataset context, MAE and RMSE will evaluate how well the RS can predict a user's rating for a movie based on a scale from one to five stars [4]. **MAE** is calculated by averaging the absolute deviation of a user's estimated rating and actual rating.

The formula for MAE is:

$$\mathrm{MAE} = \frac{\sum_i^n | r_i - e_i |}{n} \qquad [18]$$

RMSE is calculated by finding the square root of the average squared deviations of a user's estimated rating and actual rating. The formula is:

$$\text{RMSE} = \sqrt{\frac{\sum_i^n (r_i - e_i)^2}{n}} \quad [19]$$

Where in both formulas for MAE and RMSE **n** is the total number of items, **i** is the current item, $\mathbf{r}_i$ is the actual rating a user expressed for i, and $\mathbf{e}_i$ is the RS's estimated rating a user has for i.

Since RMSE squares the deviations and MAE only sums the deviations, RMSE will weight larger deviations more than MAE. In the context of movie ratings, RMSE may provide a more insightful accuracy metric but we provide both RMSE and MAE evaluation on the RS algorithms. The smaller RMSE and MAE are, the more accurate a RS. This is because RMSE and MAE will calculate smaller values if the deviations between actual and predicted ratings are smaller.

Coverage measures how many recommendations a RS is able to make for users. It is calculated by dividing the total number of preferences that a RS was able to estimate by the total number of preferences the RS attempted to estimate. In some cases, there is not enough data for a RS to provide recommendations. When this is a common case, a RS coverage will be a low value.

$$\text{Coverage} = \frac{\text{Total \# of Estimated Preferences}}{\text{Total \# of Attempted Estimated Preferences}} \quad [20]$$

2.4.3 Baseline Evaluation of Mahout RSs and Similarity Algorithms

Using the Apache Mahout library with the Movielens Dataset, these first experiments are concerned with evaluating commonly implemented RS and similarity algorithms. Specifically, the ML100K, ML1M, and ML10M are evaluated using CF RSs that utilize both the Pearson Correlation Coefficient and Log Likelihood similarity algorithms. The purpose of this experiment is to determine how an increasing data size of users, items, and user preferences affect the accuracy of a RS.

As a preface, it is interesting to note that the Netflix Prize was awarded to the team that improved recommendation accuracy, specifically RMSE, by 10 percent. The RMSE for the winning team was approximately 0.8567, which gives some perspective to the following results [21].

2.4.3 Results

The following results used all three datasets (ML100K, ML1M, ML10M), used 80 percent training data, and 20 percent evaluation data on each dataset. Tables and Charts 3.1-3.2 represent evaluation of User-User CF algorithms with both Pearson and Log Likelihood similarity algorithms. Tables and Charts 3.3-3.4 represent evaluation of Item-Item CF algorithms with both Pearson and Log Likelihood similarity algorithms.

Evaluation of User-User CF

Table 2.2 Evaluation of User-User RMSE using the Pearson Correlation and Log Likelihood similarity algorithms shows how as the user, item, and preference content in a data set increases, the RMSE improves for both similarity algorithms.

Data Set	Pearson RMSE	Log Likelihood RMSE
ML100K	1.15	1.03
ML1M	1.10	1.03
ML10M	1.09	0.96
Total Change	+0.06	+0.07

Chart 2.1 Reflects data from Table 2.2.

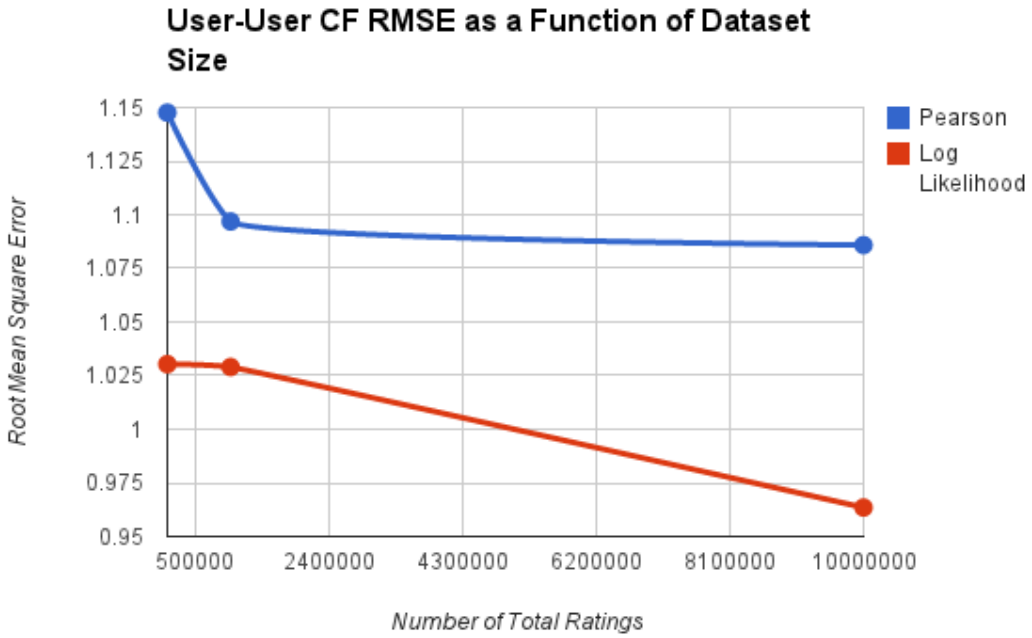
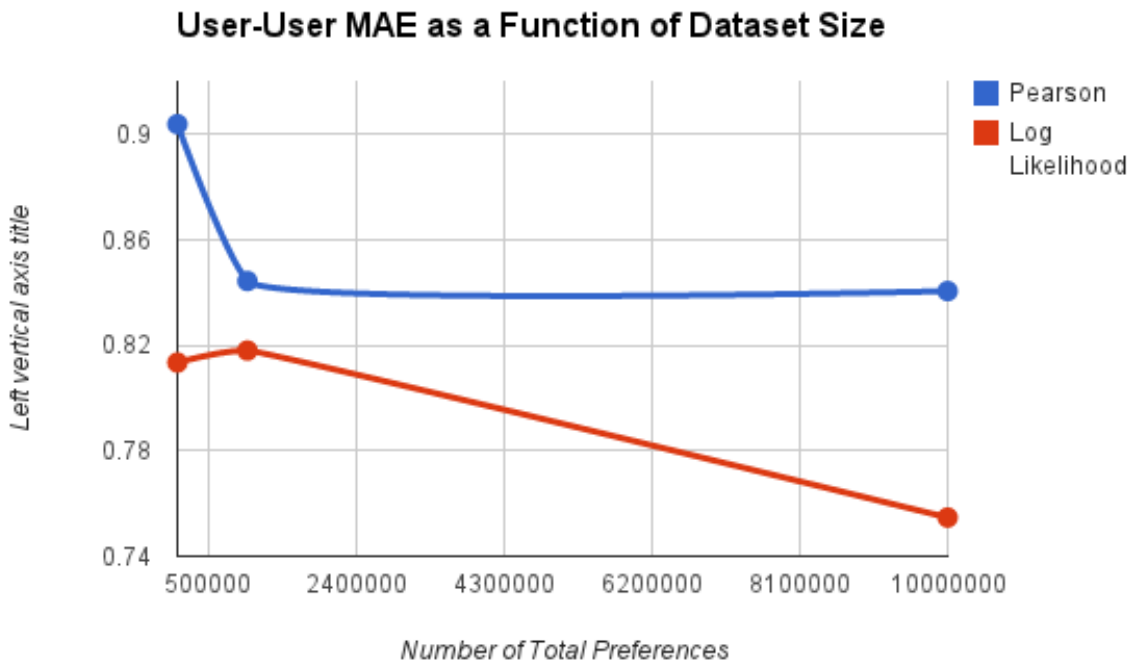


Table 2.3 Evaluation of User-User MAE using the Pearson Correlation and Log Likelihood similarity algorithms shows how, as the user, item, and preference content in a data set increases, the MAE improves for both similarity algorithms.

Data Set	Pearson MAE	Log Likelihood MAE
ML100K	0.90	0.81
ML1M	0.84	0.82
ML10M	0.840	0.75
Total Change	+0.06	+~0.059

Chart 2.2 Reflects data in Table 2.3.



Evaluation of Item-Item CF

Table 2.4 Evaluation of Item-Item RMSE using the Pearson Correlation and Log Likelihood similarity algorithms shows how, as the user, item, and preference content in a data set increases, the RMSE improves for both similarity algorithms.

Data Set	Pearson RMSE	Log Likelihood RMSE
ML100K	1.06	1.03
ML1M	1.04	1.01
ML10M	0.94	0.99
Total Change	+0.12	+0.04

Chart 2.3 Reflects data from Table 2.4.

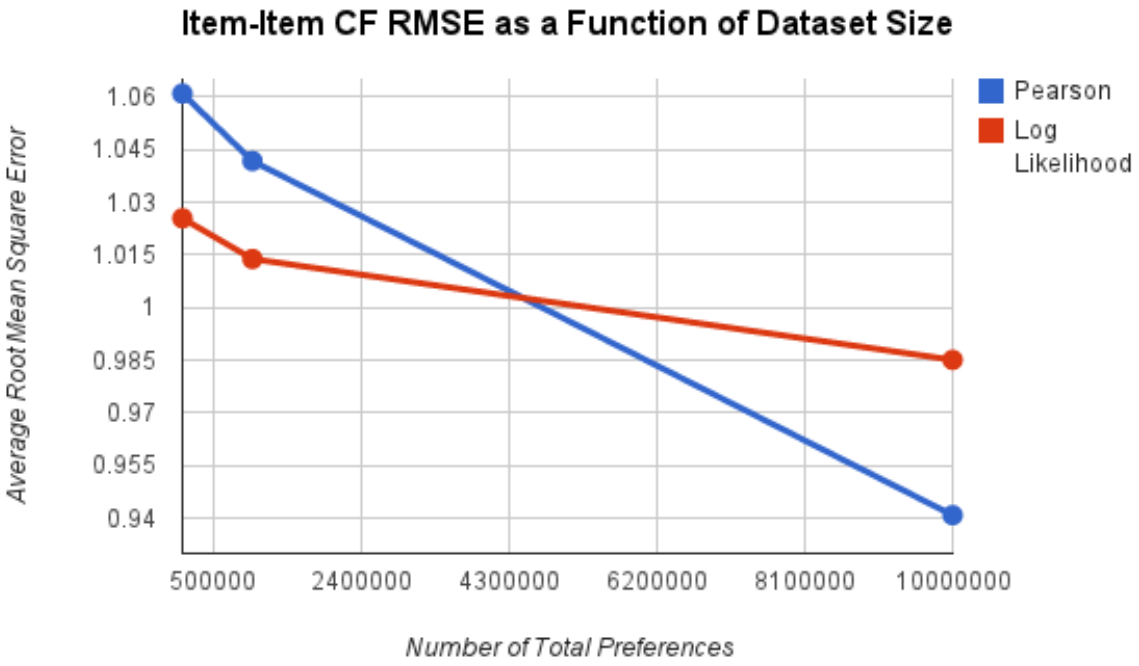


Table 2.5 Evaluation of Item-Item MAE using the Pearson Correlation and Log Likelihood similarity algorithms shows how, as the user, item, and preference content in a data set increases, the MAE improves for both similarity algorithms.

Data Set	Pearson MAE	Log Likelihood MAE
ML100K	0.83	0.82
ML1M	0.82	0.81
ML10M	0.73	0.78
Total Change	+0.10	+0.04

Chart 2.4 Reflects data in Table 2.5.

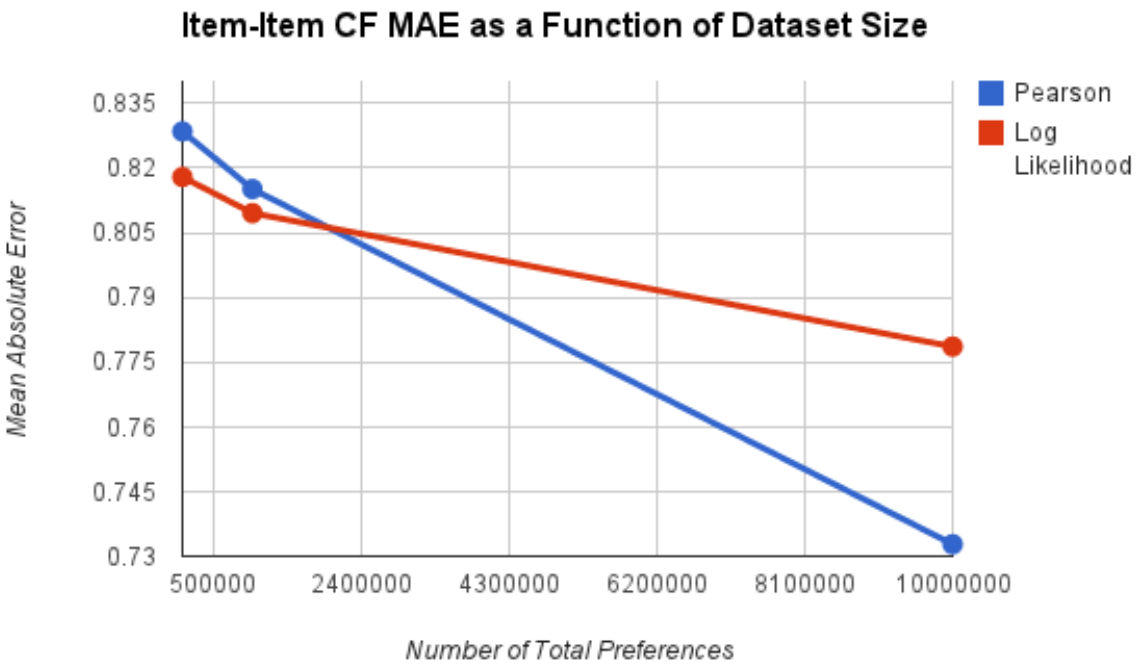


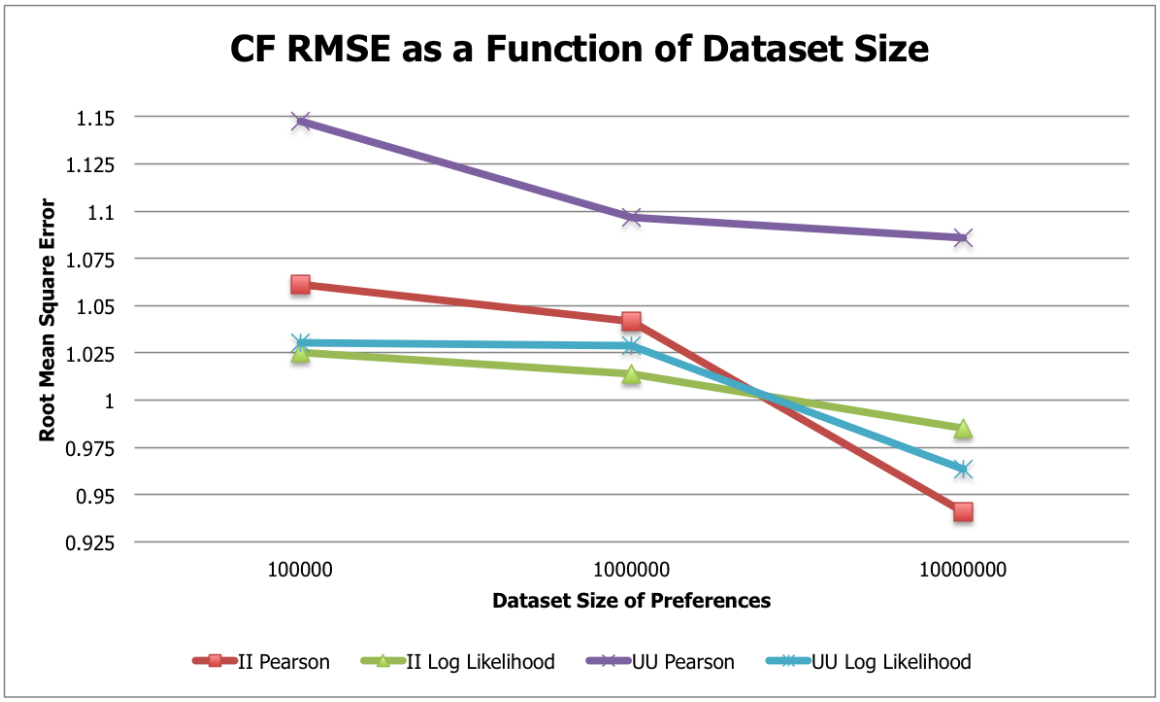
Table 2.6 As the data set size increases, the median number of ratings per user increases from ML100k to ML1M but decreases from ML1M. The mean number of ratings per item increases from ML100k to ML10M.

Data Set	Median # Ratings Per User	Median # Ratings Per Item
ML100K	106.04	59.45
ML1M	165.60	269.89
ML10M	143.11	936.60

Table 2.7 As the data set size increases, the mean number of ratings per user increases from ML100k to ML1M but decreases from ML1M. The mean number of ratings per item increases from ML100k to ML10M.

Data Set	Mean # Ratings Per User	Mean # Ratings Per Item
ML100K	65	27
ML1M	96	123.5
ML10M	69	135

Chart 2.5 This chart reflects the change in RMSE as a function of dataset size across each User-User and Item-Item CF and similarity algorithms



2.4.3 Discussion

Across all of the evaluations, the results show that an increase in the overall content of a dataset including users, items, and user preferences improves recommendation accuracy for both User-User and Item-Item CF and both Pearson and Log Likelihood similarity algorithms. For User-User CF, the improvement is attributed to the growth in users and user preferences that allow the RS to find better neighborhoods of similar users. For Item-Item CF, the improvement is not only related to the increased number of items, but also an increase in number of user preferences for each item which allows the RS to find more similar items based on these user preferences.

It is important to note the increase in data amount between datasets and the respective change in recommendation accuracy. From data set ML100K to ML1M, there are around 6.5x more users, 2x more movies, and 10x more preferences. From ML1M to ML10M there are around 12x more users, 3x more movies, and 10x more preferences. The results show that the improvement in RS accuracy is related to an increase in content, but not directly proportional to the increase in content between data set sizes.

We also found that the most improvement in accuracy occurred for Pearson Item-Item CF. Referring to *Tables 2.6 and 2.7*, the mean number of preferences per item increases by 5x from ML100K to ML10M and the median number of preferences per item increases by 15x from ML100K to ML10M. For User-User CF, there is an increase in median number of preferences per user and a decrease in mean number of preferences per user, but the ML10M contains more overall user preferences. Since there are more preferences per item, Item-Item CF can better predict users' ratings for movies since more similar items can be calculated, and explains why there is a bigger improvement in Item-Item CF than User-User CF since the latter had less significant increases in typical user behavior and a decreased average ratings per user.

These results should be taken into consideration when designing a RS. While we found that an increase in overall content of users, items, and preferences improves accuracy, there was better improvement for Item-Item CF than User-User CF, and Item-Item Pearson CF performed the best. This implies that utilizing the similarity between movies is a better method of recommendation than relying on the social and peer context of the similarity between users.

2.4.4 Recommendation Accuracy and Coverage As A Function of Target User Ratings Pool

The baseline evaluation of RSs with varying data sets proved how different CF RSs and similarity algorithms improve with more content. This experiment was designed to examine how recommendation accuracy and coverage change as users introduce new movie ratings to the RS in increments. It also simulates how RSs respond to the cold start problem, or when a RS must provide "recommendations to novel users who have no preference on any items," since the experiment does not use the target user's preferences in the training data model and adds one increment at a time in chronological order, just as a new user would be introduced to the data model, rating one movie at a time [22].

2.4.4 Pseudo Code

Pseudo Code to Evaluation RSs as a Function of Target User Ratings Pool Size

For each user **u**

Add all preferences from other users that occurred before **u**'s first preference to training data model $\mathbf{d_t}$

Get all of **u**'s preferences $\mathbf{p_u}$ in chronological order

For each preference **p** from $\mathbf{p_u}$

Add **p** to $\mathbf{d_t}$

Evaluate recommendation (using RMSE and MAE) on the evaluation data set for **u** using $\mathbf{d_t}$as the training data

2.4.4 Results

The following results used the ML1M dataset, 50 percent training data, and 50 percent evaluation for each user that rated 20 movies (which is every user in this dataset). Only 20 movies were considered, therefore the training data consisted of 10 movie ratings in increments of one movie and the evaluation was on the last 10 preferences of each user. It evaluated accuracy using both CF and similarity algorithms and recommendation coverage.

Evaluation of User-User CF

Table 2.8 As the number of preferences in the training model increases for each user, User-User CF RMSE improves for both similarity algorithms.

Number of Preferences Per User	Pearson RMSE	Log Likelihood RMSE
1	#N/A	1.18
2	1.43	1.16
3	1.34	1.15
4	1.28	1.14
5	1.25	1.14
6	1.23	1.13
7	1.22	1.13
8	1.20	1.12
9	1.20	1.12
10	1.18	1.11

Chart 2.6 Reflects data from Table 2.8.

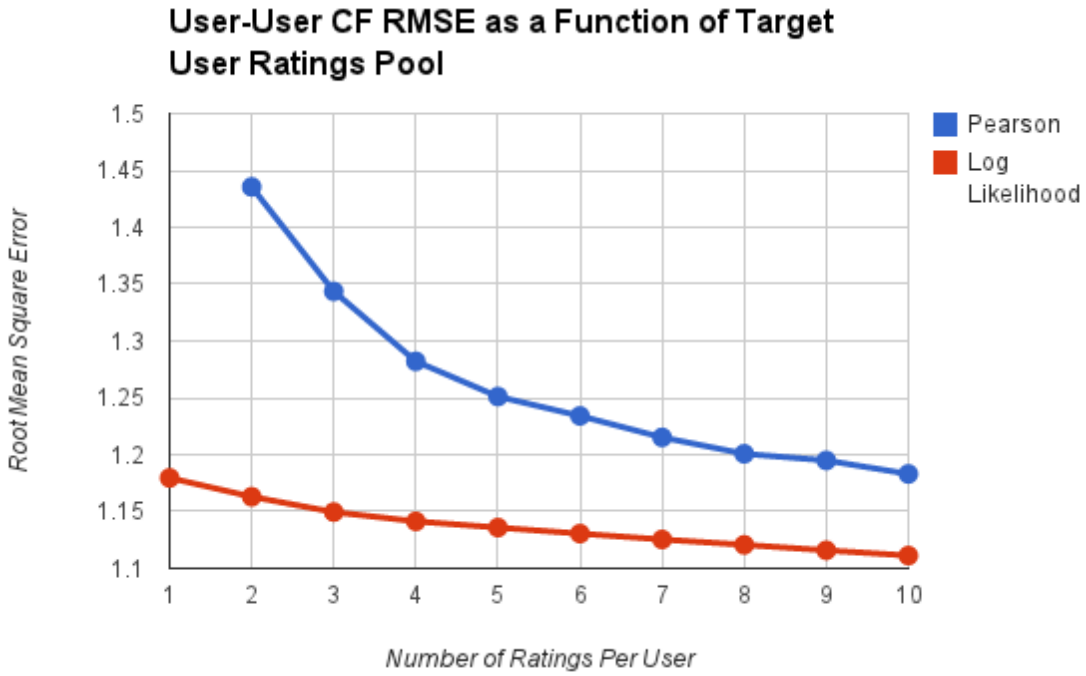


Table 2.9 As the number of preferences in the training model increases for each user, User-User CF MAE improves for both similarity algorithms.

Number of Preferences Per User	Pearson MAE	Log Likelihood MAE
1	#N/A	0.85
2	0.92	0.85
3	0.90	0.84
4	0.88	0.84
5	0.87	0.84
6	0.86	0.84
7	0.86	0.83
8	0.86	0.83
9	0.85	0.83
10	0.85	0.83

Chart 2.7 Reflects data from Table 2.9.

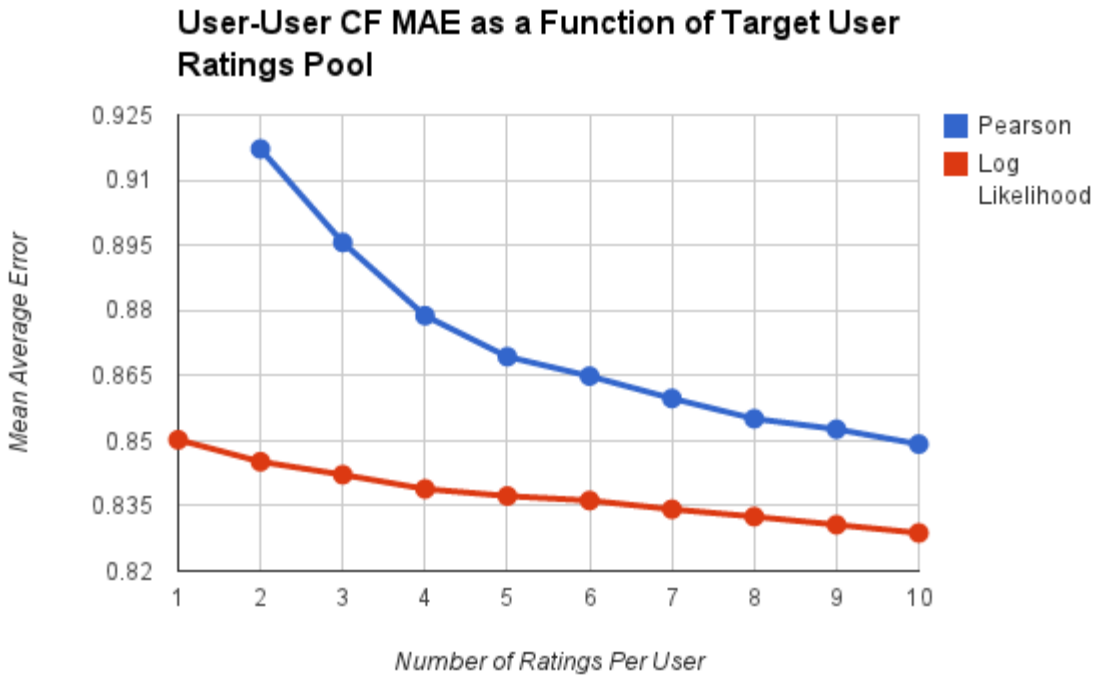


Table 2.10 As the number of preferences per user increases for User-User CF, Pearson Coverage initially increases but eventually decreases while Log Likelihood consistently increases. This slight decrease in Pearson Coverage is discussed in *Chart 2.9* and in *2.4.4 Discussion*.

Number of Preferences Per User	Pearson Coverage	Log Likelihood Coverage
1	#N/A	0.45
2	0.85	0.66
3	0.86	0.72
4	0.86	0.74
5	0.84	0.75
6	0.83	0.76
7	0.82	0.78
8	0.81	0.80
9	0.80	0.81
10	0.78	0.83

Chart 2.8 Reflects data from Table 2.10.

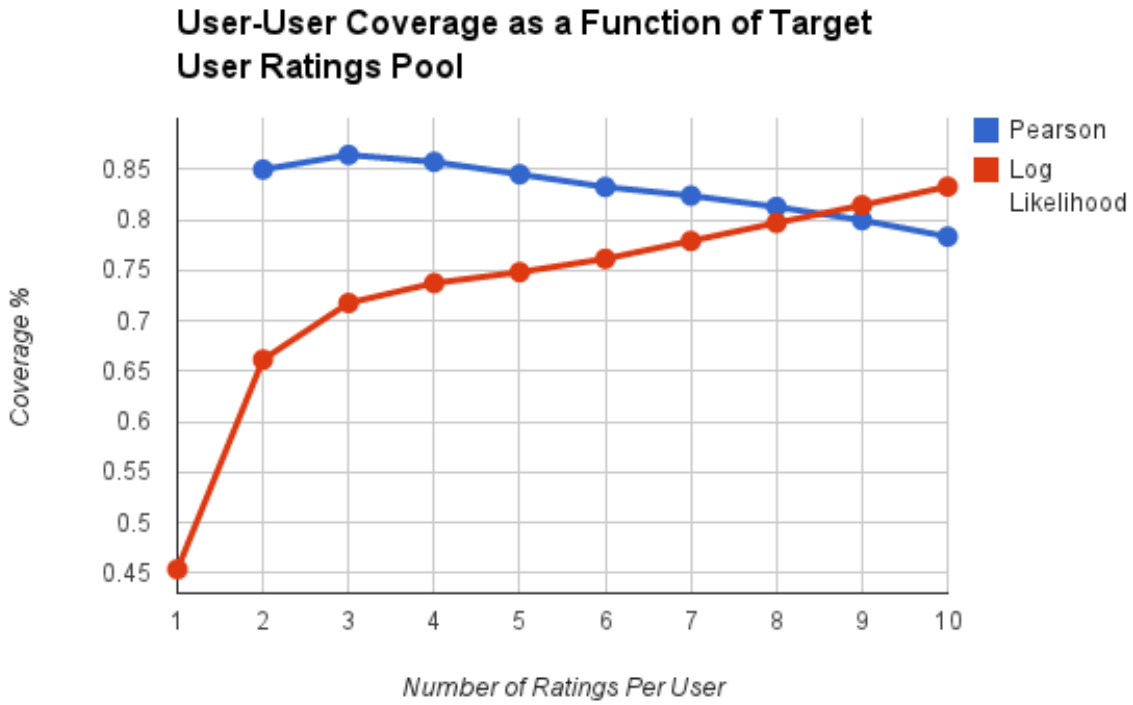
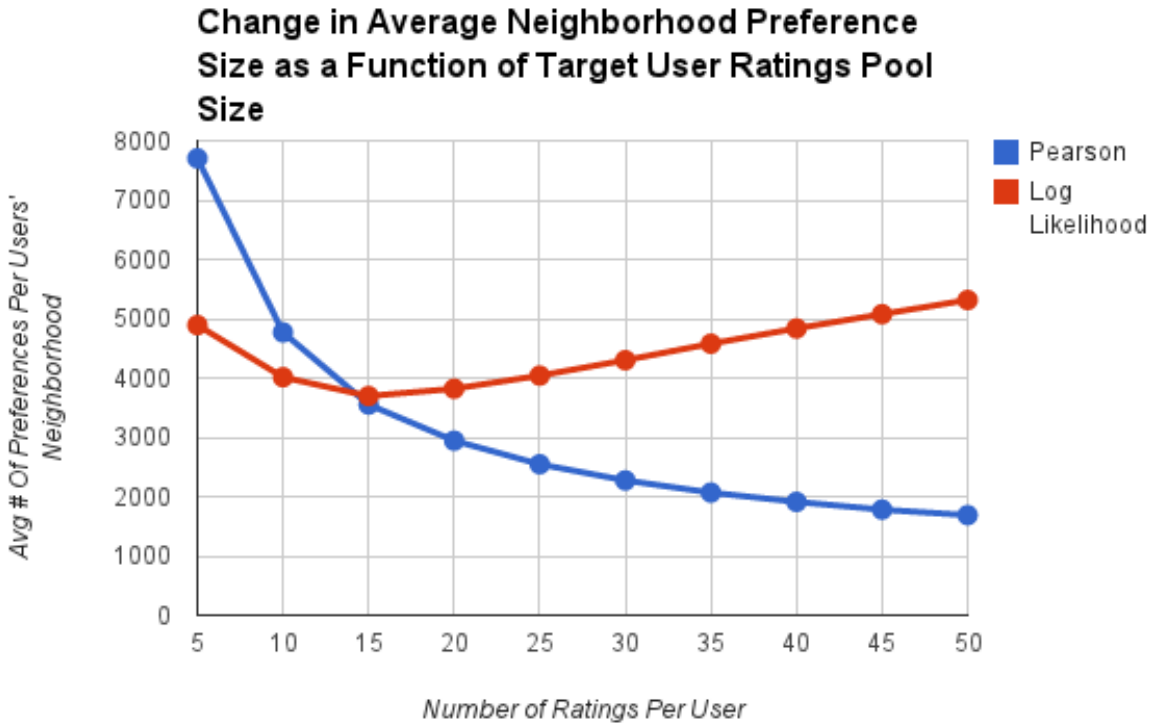


Chart 2.9 As increments of preferences for a target user are introduced to the data model, the average number of preferences for every users' neighborhood decreases for Pearson and increases for Log Likelihood similarity.



Evaluation of Item-Item CF

Table 2.11 As the number of preferences in the training model increases for each user, Item-Item CF RMSE improves for both similarity algorithms.

Number of Preferences Per User	Pearson RMSE	Log Likelihood RMSE
1	#N/A	#N/A
2	2.36	1.89
3	2.33	1.80
4	2.30	1.73
5	2.27	1.67
6	2.23	1.63
7	2.20	1.59
8	2.17	1.56
9	2.14	1.53
10	2.11	1.50

Chart 2.10 Reflects data from Table 2.11

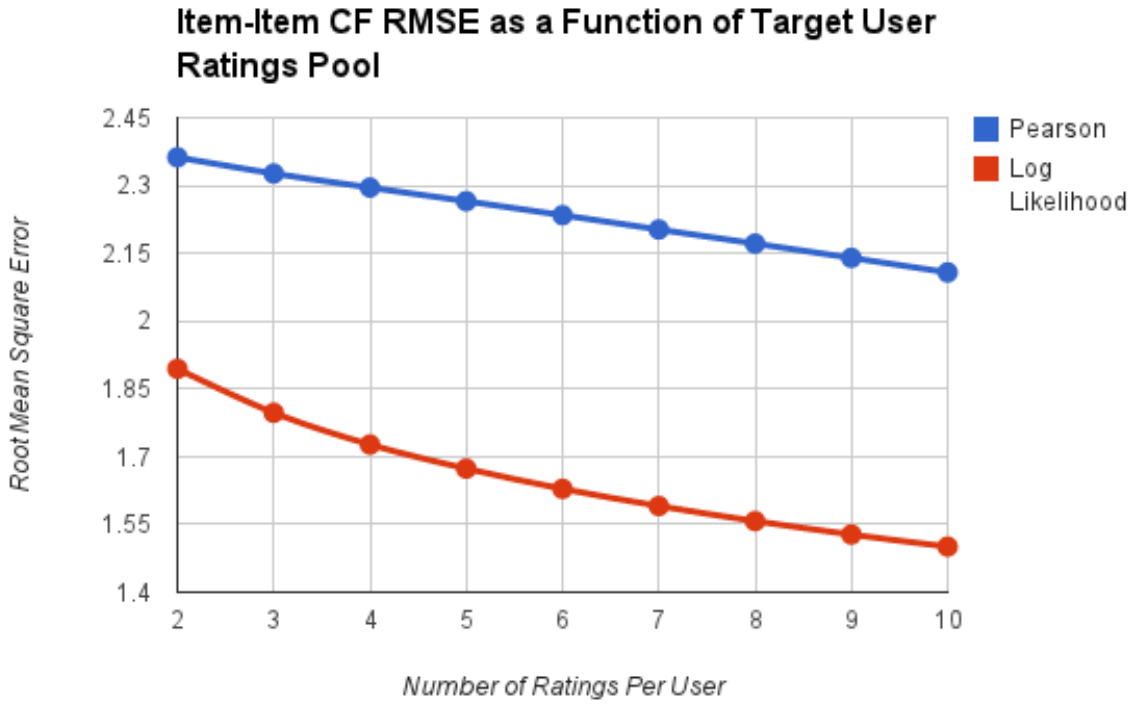


Table 2.12 As the number of preferences in the training model increases for each user, Item-Item CF MAE improves for both similarity algorithms.

Number of Preferences Per User	Pearson MAE	Log Likelihood MAE
1	#N/A	#N/A
2	1.19	1.09
3	1.18	1.07
4	1.17	1.05
5	1.17	1.04
6	1.16	1.03
7	1.15	1.02
8	1.14	1.01
9	1.13	1.00
10	1.12	0.99

Chart 2.11 Reflects data from Table 2.12.

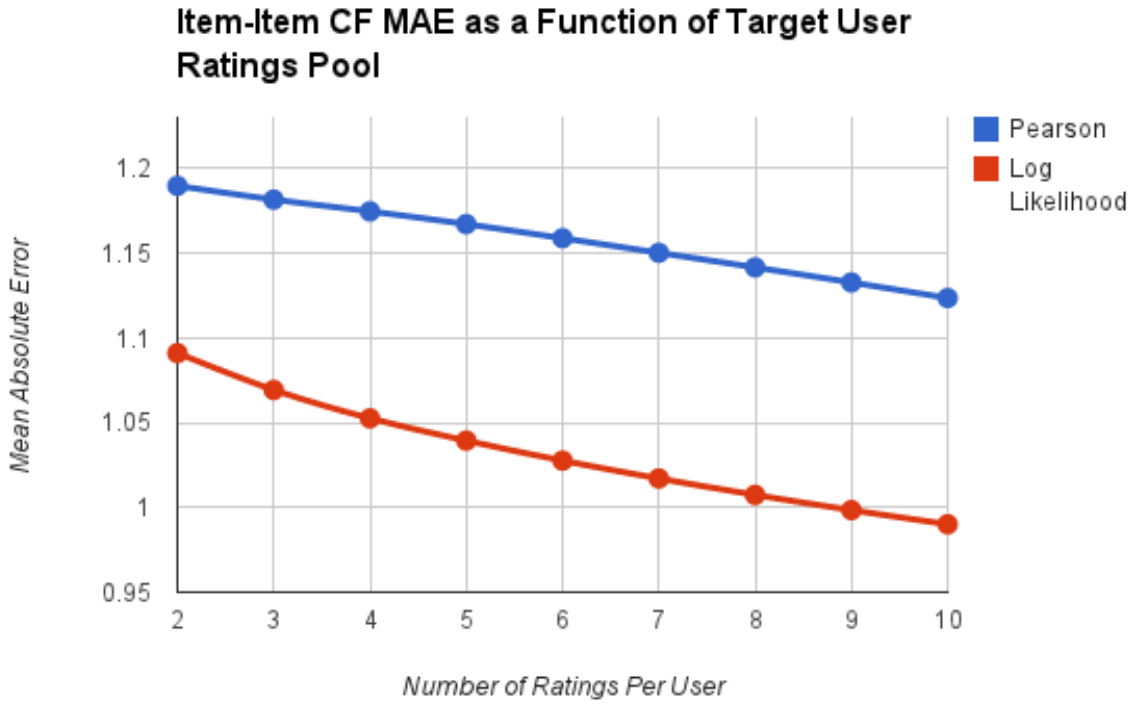


Table 2.13 As the number of preferences in the training model increases for each user, Item-Item CF coverage improves for both similarity algorithms.

Number of Preferences Per User	Pearson Coverage	Log Likelihood Coverage
1	#N/A	#N/A
2	0.95	0.98
3	0.97	0.99
4	0.98	0.99
5	0.98	0.99
6	0.98	0.99
7	0.98	0.99
8	0.99	0.99
9	0.99	0.99
10	0.99	0.99

Chart 2.11 Reflects data from Table 2.13

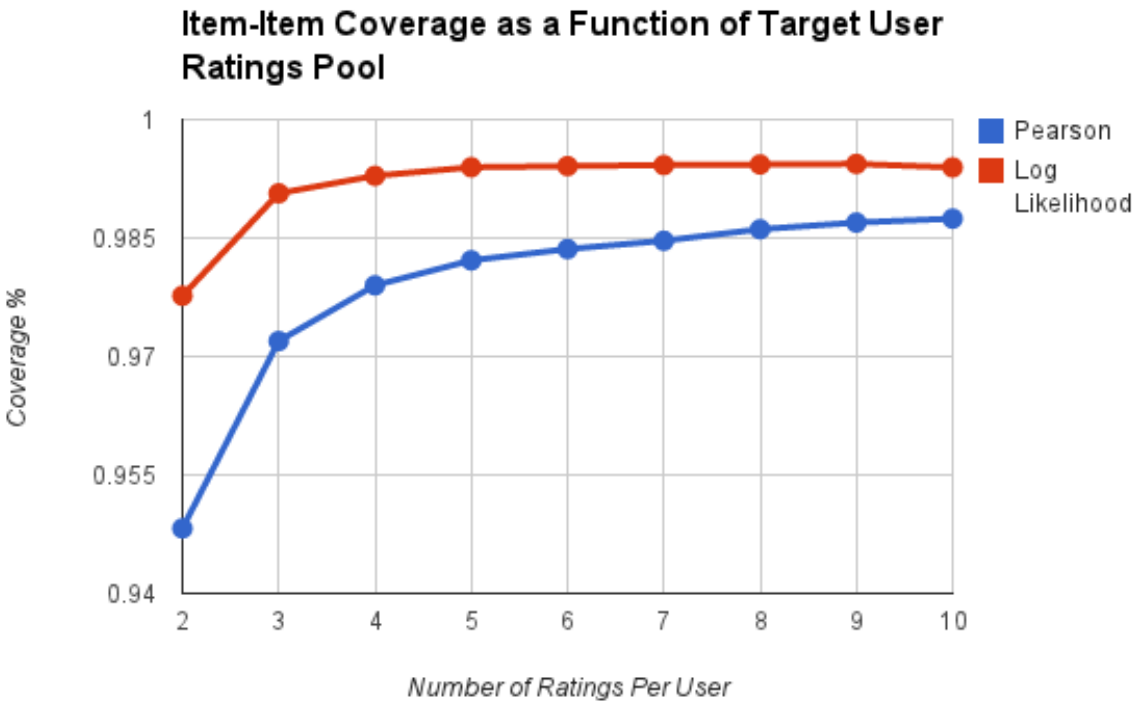
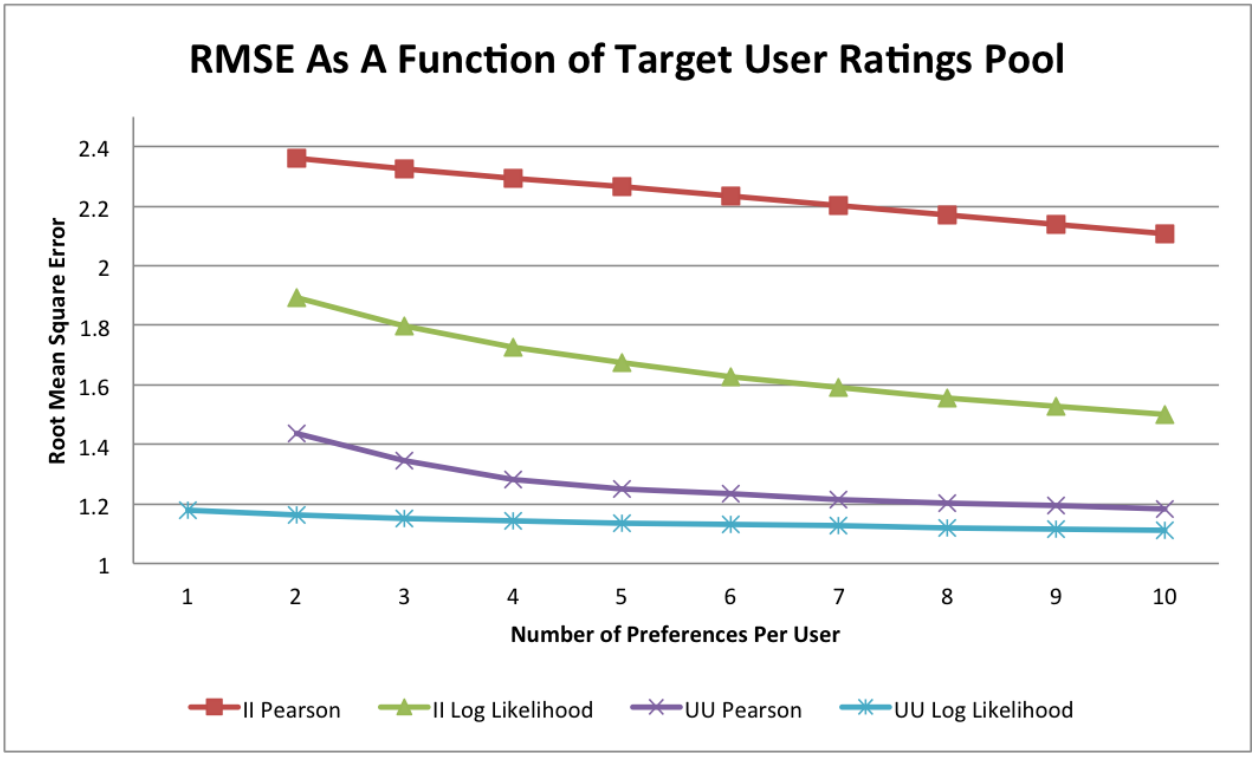


Chart 2.12 This chart reflects the RMSE as a function of target user ratings pool across each User-User and Item-Item CF and similarity algorithms. We see that UU Log Likelihood is the most accurate algorithm combination.



2.4.4 Discussion

From this experiment, we learned how a RS responds to the cold start problem and found that as a target user's preferences were incrementally added one preference at a time, recommendation accuracy improved while coverage varied according to the CF and similarity algorithm.

In User-User CF, Pearson similarity had greater improvement of recommendation accuracy but Log Likelihood performed with a better overall accuracy. Coverage slightly increased for Pearson similarity before decreasing while coverage consistently increased for Log Likelihood. This decrease in coverage for User-User CF Pearson can be explained using *Chart 2.9*. We see as each target user's preferences are incremented, the number of total preferences rated by this user's neighbors *decreases*. As a user rates more movies, while the RS creates more similar neighborhoods (reflected by an increase in accuracy), these neighbors span less preferences and therefore increase the chances of the RS not being able to predict a rating for the target user. In User-User CF, Log Likelihood similarity was able to provide recommendations for users who have only rated one movie whereas Pearson similarity is unable to make recommendations until a user has rated two movies.

In Item-Item CF, the Log Likelihood similarity algorithm improved its accuracy and was overall significantly more accurate than Pearson. In both similarity algorithms, the coverage approaches 100 percent. Neither Pearson nor Log Likelihood similarity are able to make recommendations for a user who has rated one movie. This is related to the scale of users and items and the behavior of similarity algorithms. There are many more users than items, explaining why there are enough users for User-User CF to find neighborhoods for Log Likelihood, while there may not be enough similar items for Item-Item CF to find with both similarity algorithms.

It is interesting to note that this was one of several experiments that attempted to examine the accuracy and coverage of RSs after incrementally adding preferences. In other experiments, larger increments of 5 and 10 preferences were introduced and recommendation was evaluated. For these larger increments, recommendation accuracy actually decreased. While this behavior needs further exploration beyond this thesis, some variables that should be taken into consideration are the number of preferences and the timestamps between these preferences. For example, users' or items'

preferences may change over time, changing the similarities and neighborhoods from which similarity is calculated. If a user has not rated a movie in a long time period, and then rates another movie, discrepancies in neighborhoods and similarities may arise from changing user tastes or opinions about movies.

Overall, we found that when a RS is facing the cold start problem, User-User CF provides more accurate recommendations while Item-Item CF have better recommendation coverage. When designing a RS, the trade offs between these algorithms must be taken into consideration along with how accuracy and coverage will affect user experience.

2.4.5 Recommendation Accuracy and Coverage As a Function of Non-Target User Ratings Pool

We have seen how an increase in overall data set size as well as an increase in the amount of target users' preferences improves recommendation accuracy. This next experiment examines how introducing non-target data changes recommendation accuracy. For example, if a user does not rate many or any more movies, how does recommendation accuracy change for this user if *other* users rate more movies and new movies are added to the data model? The following results help answer this question.

2.4.5 Pseudo Code

Pseudo Code to Evaluate RSs as a Function of Non-Target Ratings Pool Size

For each user u

> Add all preferences from other users that occurred before u's first preference and after u's last preference, and training percentage of u's preferences, all in random order, to training data model d_t
>
> For each increment of preferences p_i from d_t
>
> > Add p_i to d_t
> >
> > Evaluate recommendation (using RMSE and MAE) on the evaluation data set for u using d_t as the training data

2.4.5 Results

The following results used the ML1M dataset, 80 percent training data, and 20 percent evaluation on each user from the dataset.

Evaluation of User-User CF

Table 2.14 As the number of non-target preference data per user increase, User-User CF RMSE improves for both similarity algorithms.

Number of Non-Target Preference Data Per User	Pearson RMSE	Log Likelihood RMSE
100,000	1.02	1.10
200,000	1.02	1.09
300,000	0.97	1.07
400,000	0.92	1.05
500,000	0.89	1.02

Chart 2.13 Reflects data from Table 2.14.

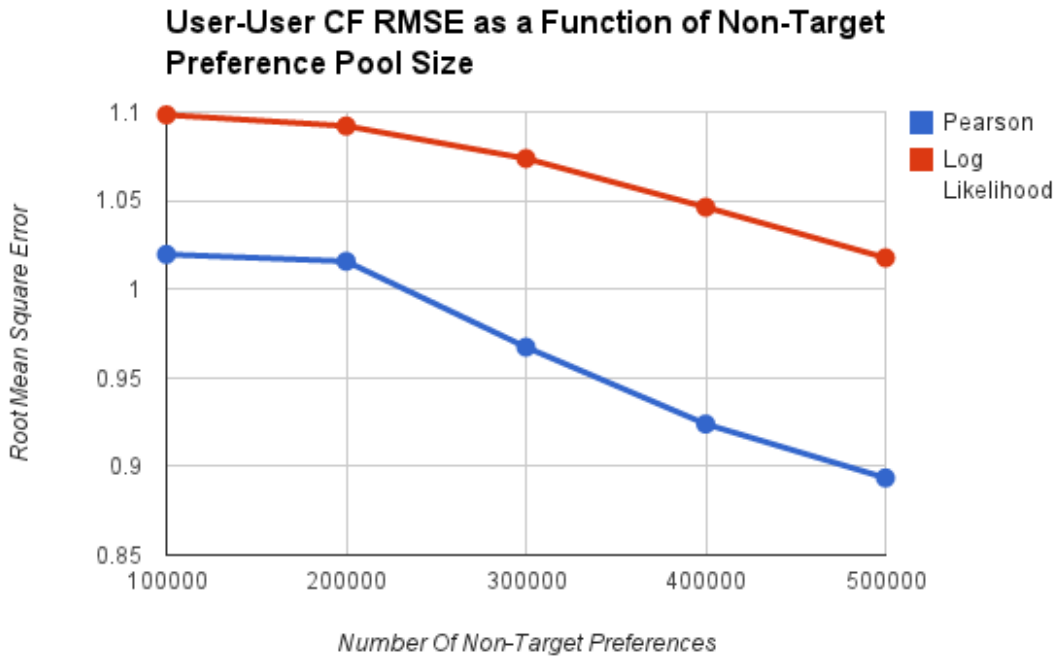


Table 2.15 As the number of non-target preference data per user increase, User-User CF MAE improves for both similarity algorithms.

Number of Non-Target Preference Data Per User	Pearson MAE	Log Likelihood MAE
100,000	0.70	0.77
200,000	0.70	0.77
300,000	0.67	0.77
400,000	0.64	0.76
500,000	0.62	0.75

Chart 2.14 Reflects data from Table 2.15.

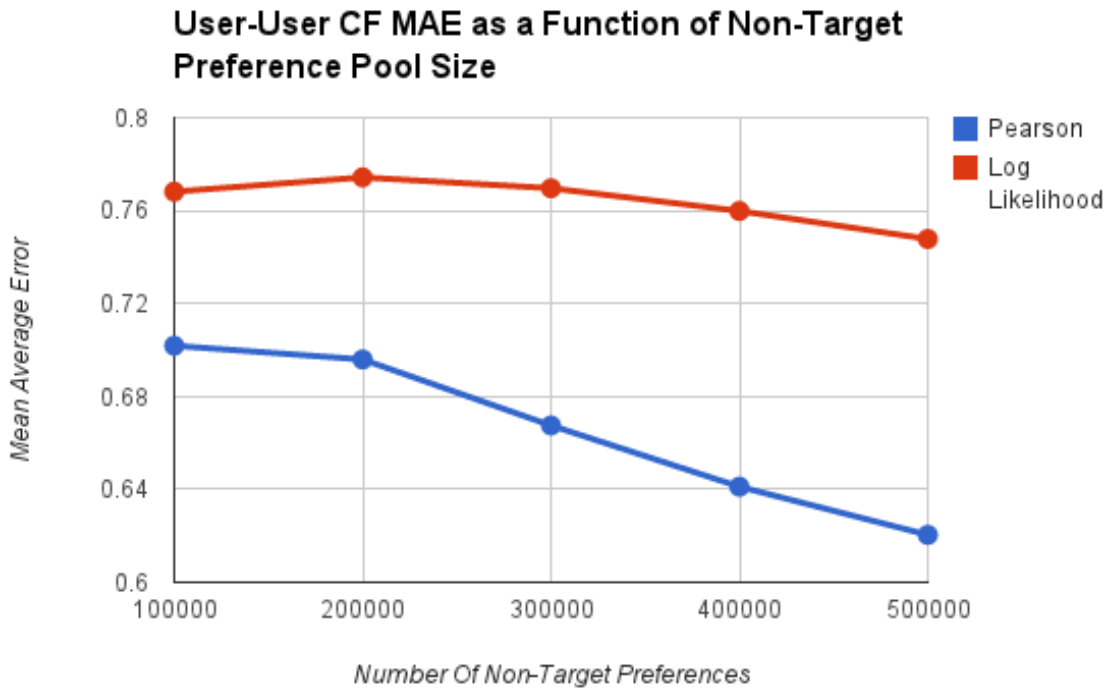


Table 2.16 As the number of non-target preference data per user increase, User-User CF coverage improves for both similarity algorithms.

Number of Non-Target Preference Data Per User	Pearson Coverage %	Log Likelihood Coverage %
100,000	0.19	0.25
200,000	0.28	0.53
300,000	0.34	0.71
400,000	0.37	0.80
500,000	0.39	0.86

Chart 2.15 Reflects data from Table 2.16.

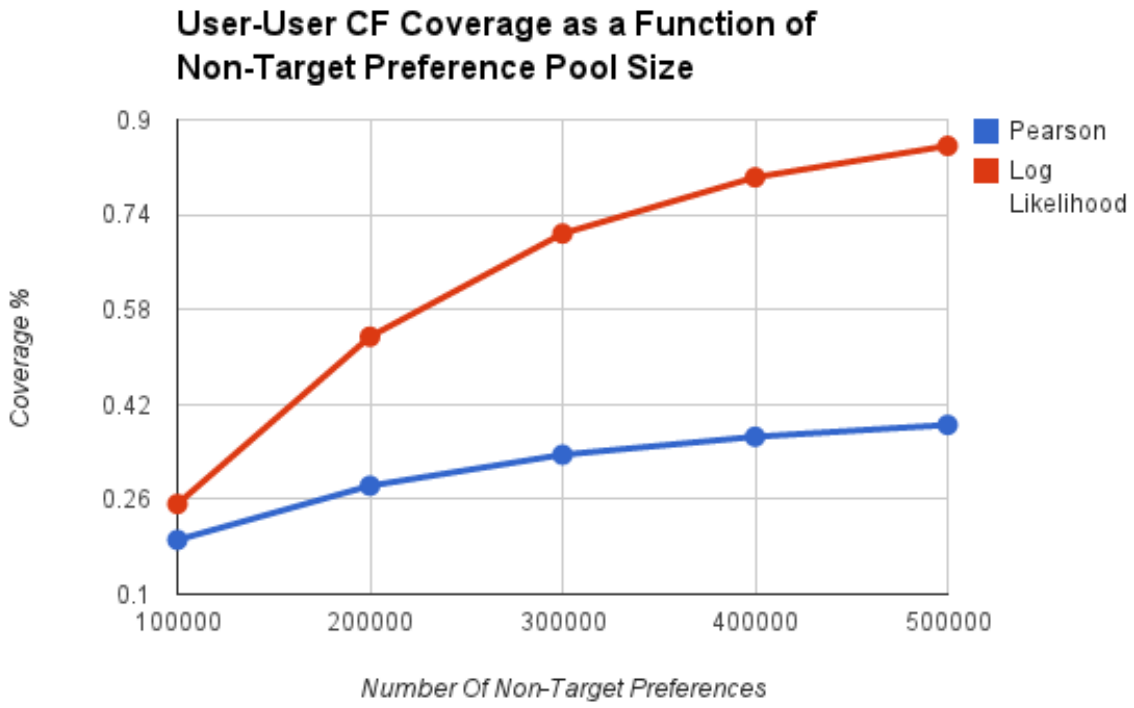


Table 2.17 As the number of non-target preference data per user increase, Item-Item CF RMSE improves for both similarity algorithms.

Number of Non-Target Preference Data Per User	Pearson RMSE	Log Likelihood RMSE
100,000	1.86	1.04
200,000	1.61	1.02
300,000	1.46	1.01
400,000	1.36	1.00
500,000	1.28	0.99

Chart 2.16 Reflects data from Table 2.17.

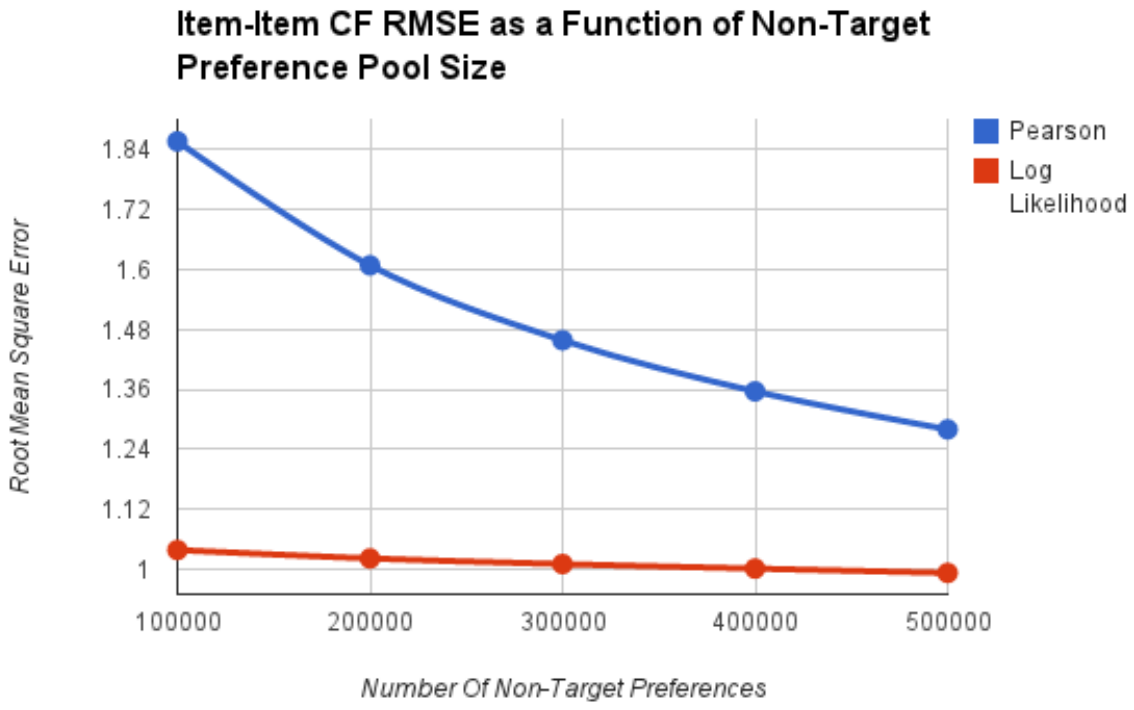


Table 2.18 As the number of non-target preference data per user increase, Item-Item CF MAE improves for both similarity algorithms.

Number of Non-Target Preference Data Per User	Pearson MAE	Log Likelihood MAE
100,000	1.02	0.80
200,000	0.95	0.79
300,000	0.90	0.78
400,000	0.87	0.77
500,000	0.84	0.77

Chart 2.17 Reflects data from Table 2.18

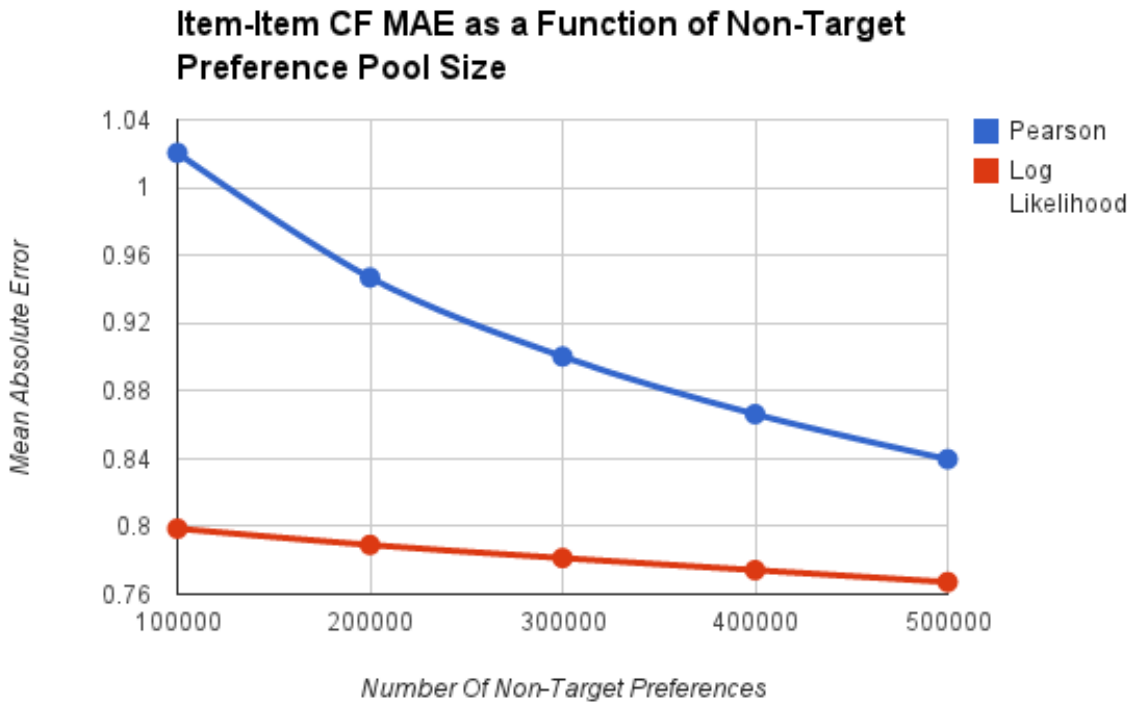


Table 2.19 As the number of non-target preference data per user increase, Item-Item CF coverage improves for both similarity algorithms.

Number of Non-Target Preference Data Per User	Pearson Coverage %	Log Likelihood Coverage %
100,000	0.951	0.995
200,000	0.992	0.999
300,000	0.997	0.999
400,000	0.999	0.999
500,000	0.999	1.000

Chart 2.18 Reflects data from Table 2.19.

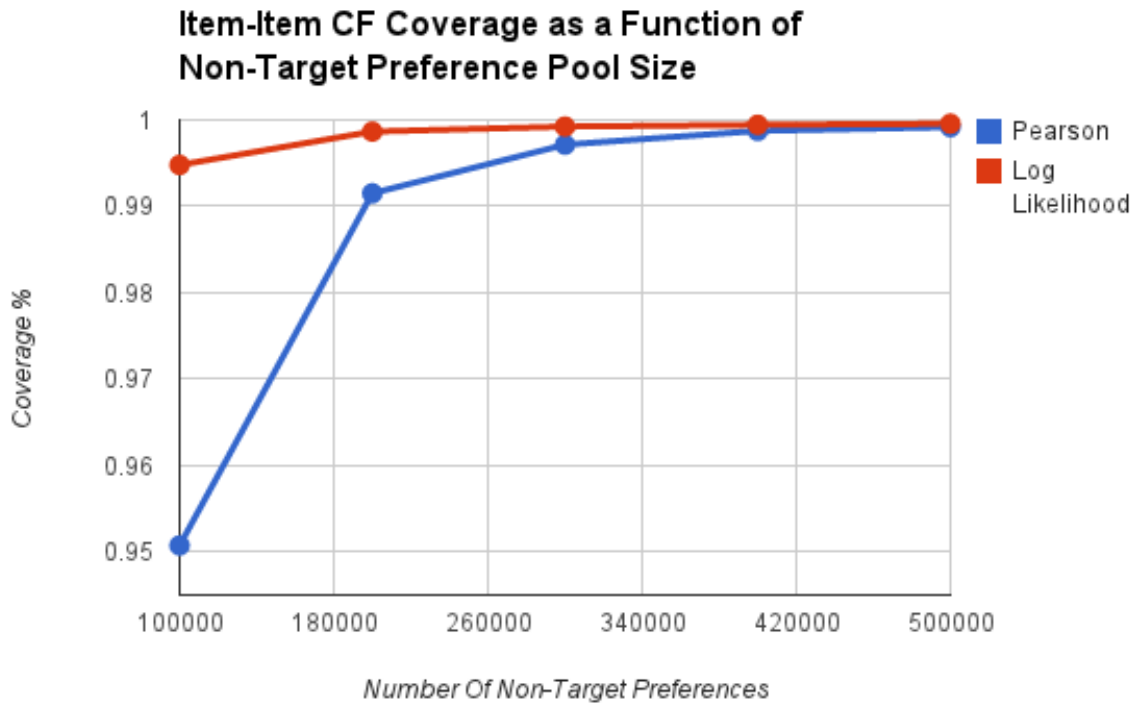
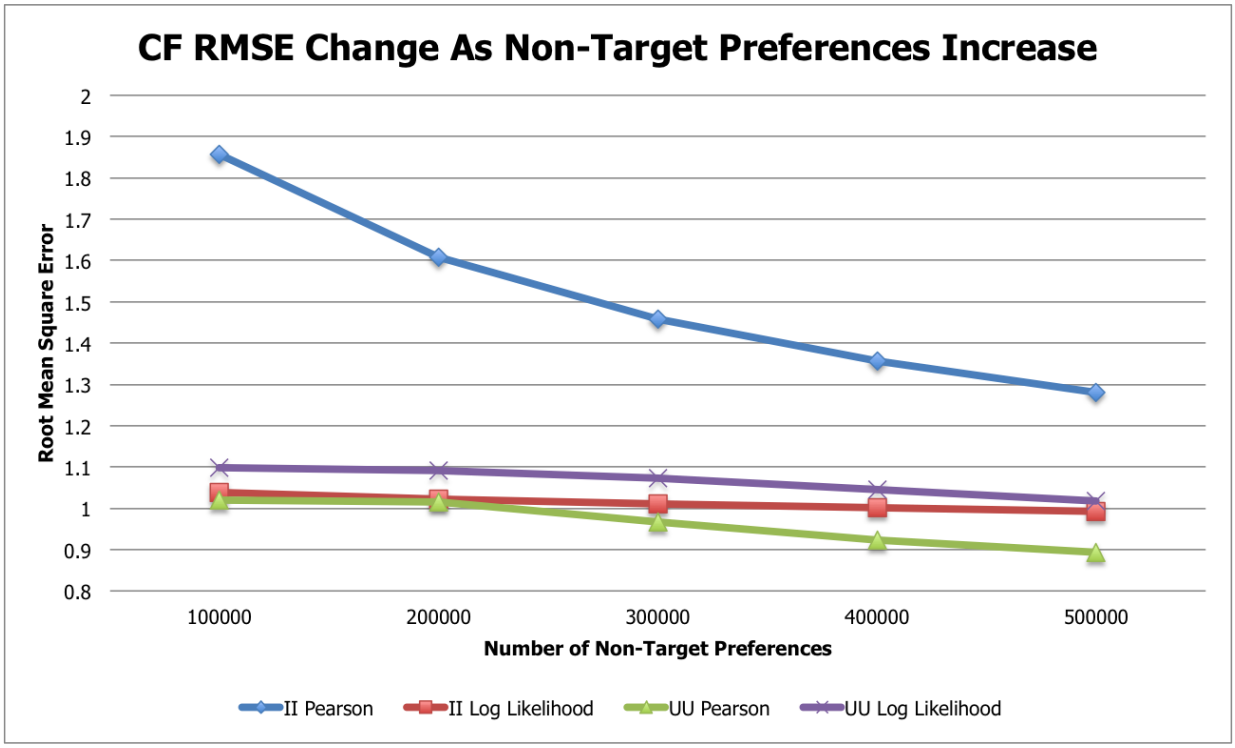


Chart 2.19 This chart shows RMSE as a function of the amount of non-target preference data across all User-User and Item-Item similarity algorithms.



2.4.5 Discussion

The results generally show that with an increase in non-target preference data, recommendation accuracy and coverage improve with both User-User and Item-Item CF and similarity algorithms. During this experiment, while each user may or may not have introduced new ratings, the focus on an increase in the "surrounding" non-target preference data introduces more users and items, allowing better user neighborhoods and item similarities to be calculated by the User-User and Item-Item CF algorithms, respectively.

In Item-Item CF we found that while Pearson had the most improvement in accuracy, overall Log Likelihood outperforms Pearson in accuracy. With both similarity algorithms, coverage improves and quickly approaches 100 percent. In User-User CF, we found that Pearson outperformed Log Likelihood in overall accuracy. Log Likelihood had considerably better coverage and approaches 90 percent, while Pearson had around half of the coverage by the time a user rates 10 movies.

User-User CF performed better than Item-Item CF with a lower RMSE and MAE, but had considerably less coverage. As more non-target data was introduced into the data model, the coverage improved with each increment, but User-User CF had considerably worse coverage than Item-Item CF's coverage. Since each increment introduced 100,000 more movie ratings, it is reasonable that overall, User-User CF with Pearson had the most accurate recommendation. With each introduction of 100,000 user preferences, it is more likely that the increase in data provides more overall similar user data than similar item data. This increase in user preferences provides better opportunities for more similar user neighborhoods to be generated by the RS. But, we still found that coverage is a problem for User-User CF algorithms and must be considered in the design and implementation of RSs.

2.4.6 Conclusions

From these experiments, we learned about which combinations of CF and similarity algorithms perform the best within the context of dataset size, accuracy, and coverage. These factors are important decisions in evaluation and designing a RS.

Larger Datasets Provide Better Recommendation Accuracy

From the results and discussion, we can conclude that a larger dataset size with more overall content including users, items, and preferences improve recommendation accuracy and coverage. In our results, Item-Item CF with Pearson similarity was the most accurate RS algorithm combination across all data sets, meaning movie similarities was a better factor in recommendation than social and peer user similarities.

Multiple RS Algorithms Must Be Used To Address The Cold Start Problem

The simulations of user increments (a new user to the data model) helped us understand how RSs behave with the cold start problem when there is varying to little availability of user preferences and similarity data. In a large dataset, when new users are introduced to the RS or a user has rated one to ten movies, User-User CF using the Log Likelihood similarity is the most accurate algorithm. However, if a user has only rated one movie, this algorithm is only able to cover around 45 percent of

recommendations. Furthermore, User-User CF with Pearson similarity is not able to make any recommendations if a user has rated only one movie. This raises questions in the implementation of RSs; some algorithms are more accurate, but may have little to no coverage. Other algorithms evaluated in the increment experiment such as Item-Item CF with both Log Likelihood and Pearson similarity achieved at least 94 percent coverage after a user rates only one movie. For RSs that are attempting to recommend to new users, our results suggest alternating algorithms according to the amount of ratings a user has submitted to the RS. Item-Item CF with Log Likelihood may be the best algorithm for users with smaller amounts of preferences while they are still building their "profile" of movie ratings. After a user has rated enough movies and the RS is able to make more recommendations, User-User CF with Log Likelihood could be utilized to make more accurate recommendations.

Increasing Surrounding Non-Target User Preference Data Improves Accuracy and Coverage

With our final experiment, we isolated a RS so that only non-target user data was incrementally added, providing a picture of how recommendation for users improves as a dataset increases in preference content around them. We found that User-User CF with Pearson similarity was the most accurate algorithm, but the Item-Item CF algorithms achieved much higher coverage. With only 100,000 movie ratings out of a large data set, it is reasonable that a RS has difficulty finding similar user neighborhoods. This allows us to conclude that in a small or growing dataset, it may be necessary to utilize Item-Item CF, despite its lower accuracy in some cases, in order to ensure that users have a positive user experience with a RS that is actually able to make movie recommendations. Perhaps users who have rated more movies, or once the data model reaches a certain size, the RS can utilize the more accurate User-User CF algorithms. It is important to consider our previous results in overall dataset size that showed Item-Item CF performs the best with larger dataset sizes. This experiment is specific to smaller dataset sizes where non-target "surrounding" preference content is increasing.

Evaluation Is Only Part Of The Bigger Picture

These results raise critical issues of evaluating a RS but do not necessarily provide a realistic evaluation of implementing a RS in production. As a RS's dataset grows and new users join the system and add new movie ratings, constraints such as speed and scale need to be considered. Which algorithms are able to provide recommendations while not interrupting the user experience with slow responses? How can we provide accurate and helpful recommendations while still meeting this requirement of a positive user experience? This next section discusses these constraints of implementing a RS as a web service.

3 IMPLEMENTING A RECOMMENDER SYSTEM AS A WEB SERVICE

Because the Apache Mahout project is a RS library, it can be used within other applications. This section of the thesis explores exposing Mahout as a web service using commonly implemented concepts including Service Oriented Architecture and RESTful Web Services.

3.1 Service Oriented Architecture

Service Oriented Architecture (SOA) is a method of organizing software systems so that there are an "interconnected set of services" that are accessible and able to communicate through "standard interfaces and messaging protocols". SOA is commonly implemented with a Service Provider that readily provides services and a Service Client that make requests to a service provided by a Service Provider. There is also a Service Registry that provides a description of the available services from a Service Provider to a Service Client. SOA focuses on designing services that maintain properties such as interoperability and loose coupling so that other applications can easily communicate with these services over common interfaces, regardless of the client's implementing technologies, while remaining decoupled from each other so that

the service provider and service client do not need to understand what the other is doing [23].

3.2 RESTful Web Services

A web service is a modular application, or Service Provider, that provides an interface by which Service Clients can retrieve application data through requests that are made through the Web [24]. REST, which stands for Representational State Transfer, is a network based application architectural style that places constraints on how elements of data can be requested and provided [25]. Therefore, a RESTful Web Service exposes web services that are available according to constraints imposed by a RESTful interface. We focus on exploring and implementing three of the main constraints of a RESTful interface including a client-server relationship, stateless communication, and a uniform interface.

3.2.1 Client-Server

The client-server constraint is based on a system architecture that is composed of two components: a server and a client. The server exposes services by listening to requests while clients make requests to this server to access these services. The server then responds accordingly, perhaps with the client's desired data from the web service or an error message if the request could not be completed [24, 25]. This architecture is a powerful constraint on RESTful Web Services because it decouples the server logic from the client; the server can focus on scaling independently and communicating with back end services such as databases while the client can focus on the user interface and application aesthetics.

3.2.1 Stateless Communication

The stateless constraint on communication between clients and servers guarantees that every request made by the client contains all of the application session state data, such as user information, required for the server to process the request and

respond. The server does not contain any context of this session state data, requiring the client to be responsible for maintaining state. This constraint on RESTful Web Services inherently provides "properties of visibility, reliability, and scalability" [24]. Every request provides all of the necessary data for the server's service to process the response and make a request often without the need of additional data retrieved internally by the service through other services. Each request ensures a high level of reliability since the design ensures all necessary request information is provided by each single request. For example, if there is a web service failure during an interaction between the client and server, the system knows that it can repeat the request since all of the data is encapsulated by the request. Lastly, a RESTful Web Service is independent of application state data, meaning it is not responsible for maintaining state across requests. The web service logic can assume that before and after every request, it does not need to concern itself with the context and state of past requests. This allows the server to "free resources" between requests and also provides for simpler distributed capabilities, since state would not need to be distributed [24, 25].

3.2.2 Uniform Interface

Lastly, the interactions between clients and servers in a RESTful web service occur over a uniform interface. This uniform interface revolves around components known as Uniform Resource Identifiers (URIs), Resources, Representations, and Hypertext Constraint that are involved in the communication process between a client and a server. A URI "is an identifier of a resource" to which a client can send requests and expect a response indicating the request's status. The Resource is the service with which the client is attempting to communicate. A Representation is an "encapsulation of the information (state, data, or markup) of the resource" such as a JSON request or response that contains resource information. Clients communicate with servers by using the Hypertext Transfer Protocol (HTTP) to submit requests to Resources and expect Representations as responses. Lastly, each response from a server to a client "represents the state of the [client's] interaction within the application". This concept, known as the Hypermedia Constraint, implies the server's response causes an altered client state and is indicative of the lack of state data in a request and response [26].

3.3 Web Service Environment

In order to build an environment in which a RS can be exposed as a web service, this thesis leveraged popular open source technologies including Java, Grizzly, Jersey's implementation of JAX-RS, Codahale Metrics, SLF4J, Apache Mahout, and MySQL (*see Appendix A for descriptions of these technologies*).

Within a Grizzly container, we use Jersey to provide RESTful web services in resource classes that are able to provide recommendations using the Apache Mahout library and underlying MySQL database with the MovieLens dataset [17] to recommend movies from a MySQL database (*see Figure 3.1*).

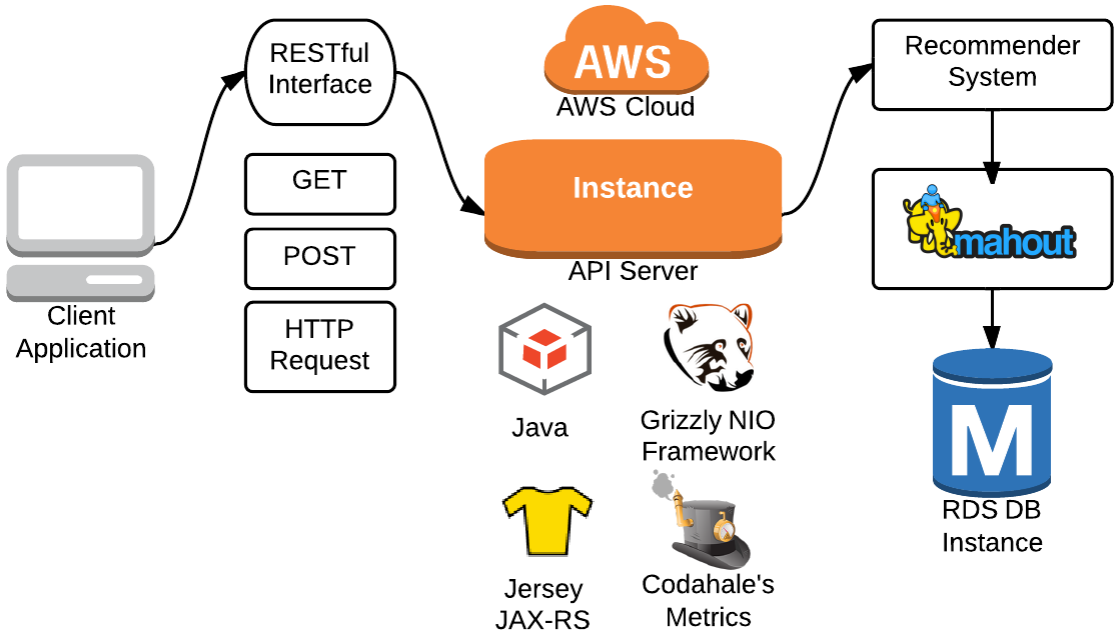


Figure 3.1 In the diagram above, we see the process of a client making an HTTP request to a web service through a RESTful interface. The web service uses a recommender system that leverages Apache Mahout and a MySQL database instance.

3.4 Experiments, Results, and Discussion

The following experiment uses the architecture described in Section 4.1. The web service application was run on an Amazon C3.Large Elastic Compute Cloud and uses an Amazon DB.M3.Medium Relational Database with MySQL.

3.4.1 Improving Recommendation Response Time

In order to simulate a web service that is handling requests from users using a movie recommendation application, we used Apache JMeter to simulate 1,000 users by making HTTP requests every second. JMeter ramps up users incrementally, so every second a new user makes an HTTP request until 1,000 users have made requests for movie recommendations. The resources available as web services are shown in *Table 3.1*.

Table 3.1 Resource URI's that represent web services that provide recommendations using varying Mahout API's.

Resource URI
/ii/file/similarity/pearson
/ii/file/similarity/loglikelihood
/ii/db/similarity/pearson
/ii/db/similarity/loglikelihood
/uu/file/similarity/pearson
/uu/file/similarity/loglikelihood
/uu/db/similarity/pearson
/uu/db/similarity/loglikelihood

Each URI represents a different manner by which recommendations are made. All URI's that contain *file* use a file as a data model that contains user preferences and uses standard Mahout API's to make recommendations. All URI's that contain *db* use a database as the data model but also leverage efficient Mahout caching API's that cache user neighborhoods, item similarities, and item-item and user-user recommenders. These web services focus on bringing recommendation data such as user preferences in memory for faster recommendation and responses to HTTP requests. One API in particular reloads data from a database into memory and also allows the data model to be refreshed so new preference data can be taken into consideration.

3.4.1 Results

Response Times For 1,000 Unique Clients

Table 3.2 Shows the performance of algorithms with varying data models. All algorithms labeled with DB used a database as a data model and also utilized Mahout's API's. All algorithms labeled with File used a File as a data model and did not utilize Mahout's caching API's.

RS Algorithm and Data Model	Median Response Time (Milliseconds)	Mean Response Time (Milliseconds)
UU DB Pearson	76	79
UU DB Log Likelihood	215	250
II DB Pearson	3434	5614
II DB Log Likelihood	3691	5888
UU File Pearson	4325	4470
UU File Log Likelihood	5000	5195
II File Pearson	18082	13835
II File Log Likelihood	11582	15161

Chart 3.1 Reflects the Median Response time in Table 3.2.

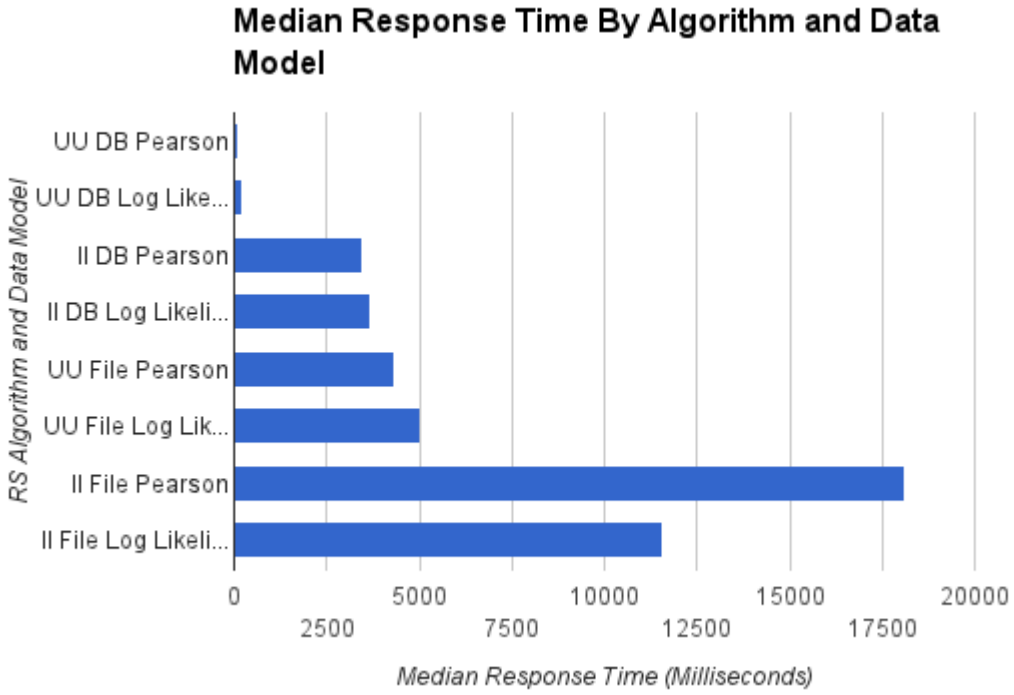
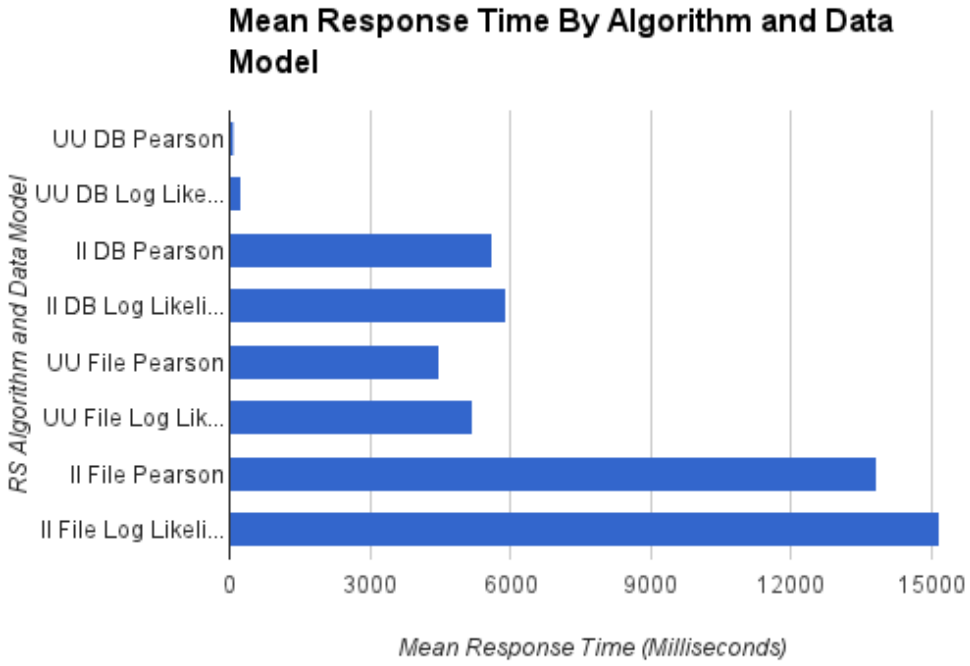


Chart 3.2 Reflects the Mean Response time in Table 3.2.



Response Times For One Unique Client

Table 3.3 Represents the Median and Mean Response times for one unique client that makes 1,000 HTTP requests to a web service client.

RS Algorithm	Median Response Time (Milliseconds)	Mean Response Time (Milliseconds)
UU Pearson	43	45
UU Log Likelihood	22	25
II Pearson	19	49
II Log Likelihood	17	42

Chart 3.3 Reflects Median Response times from Table 3.3

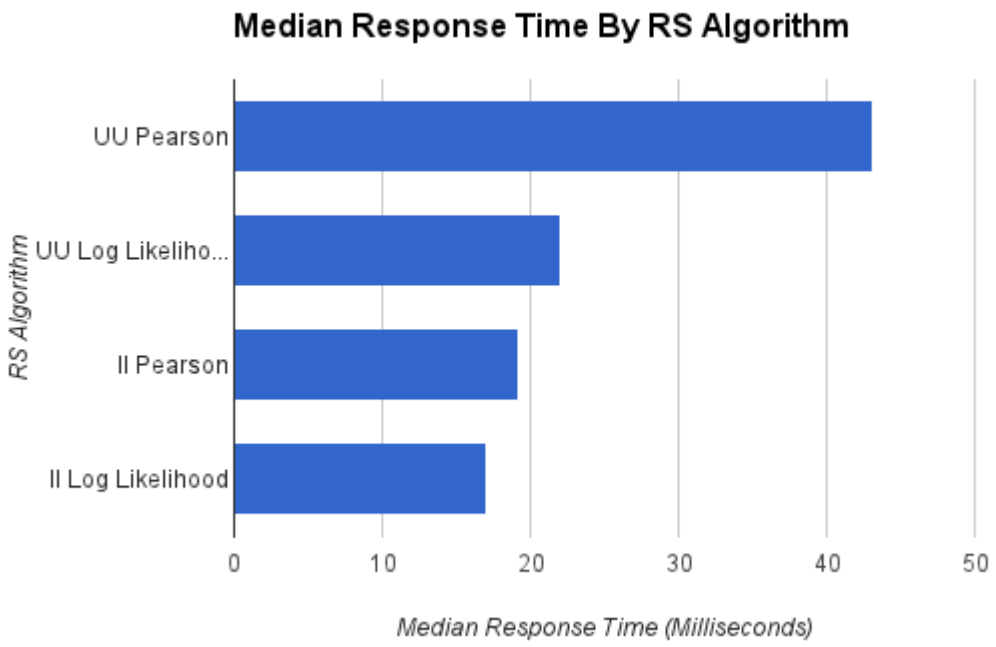
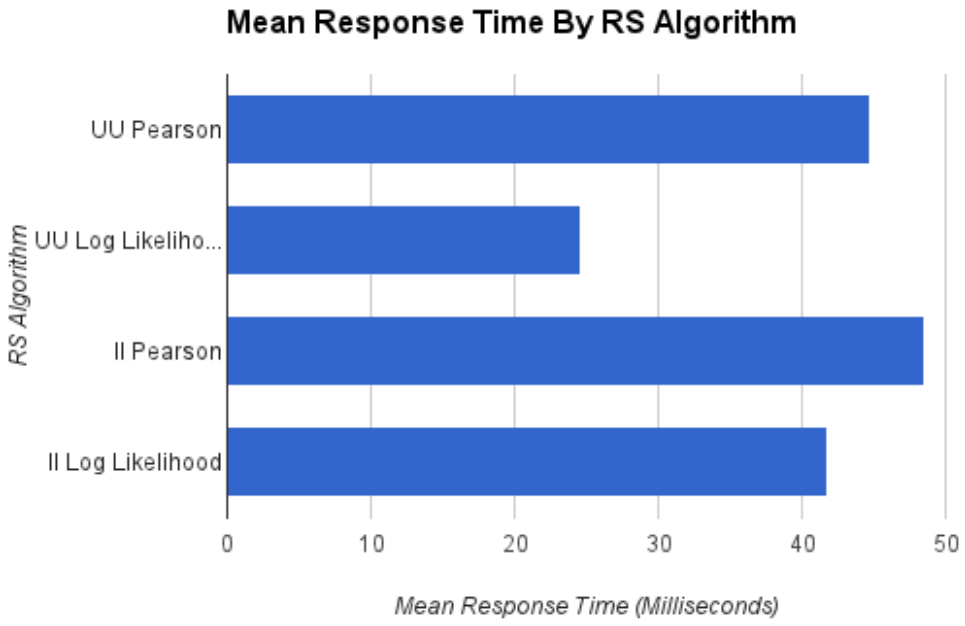


Chart 3.4 Reflects Mean Response times from Table 3.3.



3.4.1 Discussion

In our first experiment, we found that the typical response time for User-User DB Pearson was the fastest while the typical response time for Item-Item File Pearson was the slowest. In the second experiment, however, we found that both Item-Item algorithms provided the fastest recommendation response times. In the second experiment, User-User DB Pearson had double the response times of other Item-Item algorithms.

Our first simulation of 1,000 unique clients requesting recommendations represents the behavior of a RS that is being accessed by many different users at once. In some cases this may be unrealistic since not all, or many different, users would be logged into a movie recommendation service at once. The second experiment simulates the repeated HTTP requests of one client. For example, a user may log into a movie recommendation web site and browse through the recommendations. This simulation shows how caching this data for one user results in much more acceptable response times as fast as around 17 milliseconds.

3.4.2 Conclusions

By utilizing Mahout API's, the response times across all algorithms were significantly improved. In some cases, such as Item-Item Pearson, response time improved by around 20 percent. The Mahout API's implemented in algorithms labeled with DB focused on caching data in memory for fast access. For example, we utilized API's that provided a cached data model from a MySQL database, cached user and item similarity algorithms such as Pearson and Log Likelihood, cached user neighborhoods, cached item similarities, and even cached RSs. While recommendation accuracy is important to the success and user experience of a RS, if the recommendations take too long to be presented to a user, it can negatively affect a product or service. Therefore, it is important to cache data or provide faster access to data than repeated requests to a file or database storage of preference data.

4 FUTURE WORK

4. Precomputing Item Similarities and Storing in Database

While Item-Item CF algorithms ran slower than User-User CF algorithms in Experiment 4.3.1, Item-Item CF algorithms can be faster for data sets that have a much larger amount of users than items. Furthermore, the item similarities between items can be precomputed and stored in a database. This would allow more systems to easily access these precomputations when needed, rather than using memory.

4.1 Improving CF Algorithms

The Apache Mahout library is powerful because it provides baseline algorithms for effective CF algorithms. It also provides a framework and easy methods to introduce new recommendation algorithms. While exploring the Mahout APIs, we experimented with an Item Average Recommender that produces Top-N item recommendations based on overall rating averages across all items. In a simple experiment, we rescored items based on how many total ratings each item received. For example, if an item received an average rating of five stars, but only had two total ratings, this item would be "penalized" according to a pessimistic constant value. Items that may have a lower average, such as four stars, but have over 100 ratings, will not be penalized as much. In some experiments, this algorithm performed significantly better than the Item Average Recommender implemented in Mahout.

4.2 Utilize Scalability of The Cloud

While the web service prototype we present utilizes important architectural design decisions such as a RESTful interface, it does not offer true scaling capabilities. In order to utilize the scalability of the cloud, a RS web service would implement load balancing and have multiple EC2 and RDS instances that would be listening for and serving HTTP requests. This would improve web service performance, provide faster recommendations, and allow the RS to scale with an increasing number of users.

4.3 Distributed Recommender Systems

The Apache Mahout library uses algorithms implemented by a popular open source project called Apache Hadoop. Hadoop is "software for reliable, scalable, distributed computing" that uses, for example, thousands of machines to complete computation. The benefit is that each machine uses its own local resources to complete these computations and is synchronized at the application level [27]. This could dramatically reduce computation times of tasks such as precomputing item-item similarities.

In April, the Mahout project announced that it would not be accepting future algorithm implementations that utilize MapReduce, Hadoop's "system for parallel processing of large data sets" [27]. Instead, Mahout will rely on a domain specific language for "linear algebraic operations" that will run on Apache Spark, another open source project that is an "engine for large-scale data processing" [28, 29].

A WEB SERVICE IMPLEMENTATION APPENDIX

A.1 Codahale Metrics Instrumentation

When evaluating the performance of a web service, it is important to have a way of measuring metrics such as response time. We used a Java library called Codahale Metrics that "provides a powerful toolkit of ways to measure the behavior of critical components in your production environment" [30]. Specifically, we used a metric unit called a timer that measured response times each type of RS and similarity algorithm combination.

A.2 Logging

An important part of a RS prototype is logging debugging, error, and general information about the system. While logging provides various ways of monitoring the health of a system, it is also critical to instrumenting the code base and providing important data from which efficiency and behavior can be extracted. In order to

interface with other libraries such as Codahale Metrics for instrumentation, two logging libraries are used in the prototype: Simple Logging Facade for Java (SLF4J) and Log4J.

SLF4J provides a facade design pattern that abstracts common logging frameworks. Different libraries utilize different logging frameworks such as java.util.logging or Log4J. SLF4J allows a developer to decide on the underlying logging framework at deployment while easily integrating with other libraries. This allows a developer to be unbound by library dependencies on multiple logging frameworks [31].

In the RS implementation, Apache's Log4j 2 is used to log important events in the RS's lifecycle as well as a way to record the instrumentation information provided by the Codahale Metrics library. Using XML, a system can configure default and class level logging behavior as well as establish appenders that record logging information. In this prototype, logging is recorded in text files [32].

A.3 Grizzly

Grizzly is a Java framework that helps developers build scalable server applications that take advantage of powerful Java API's such as NIO [33].

A.4 MySQL

MySQL is an "Open Source SQL database management system, is developed, distributed, and supported by Oracle Corporation". It provides a database management system with relational database structures and is "very fast, reliable, scalable, and easy to use" [34].

References

[1] N. Rastin and M. Zolghadri Jahromi, "Using content features to enhance performance of user-based collaborative filtering performance of user-based collaborative filtering," *Int. journal of artificial intelligence and applications*, vol. 5, no. 1, pp. 53-62, Jan, 2014.

[2] F. Ricci et al, "Introduction to Recommender Systems Handbook," in Recommender Systems Handbook. New York: Springer, 2011, pp. 1-35.

[3] J. Konstan and M. Ekstrand. (2014, September 3). "Introduction to Recommender Systems: Module 1-8" [Online lecture]. Available: https://www.coursera.org/course/recsys

[4] J. Herlocker et al, "Evaluating collaborative filtering recommender systems, " *ACM Transactions on Information Systems (TOIS)*, vol. 22, no. 1, pp. 5-53, Jan, 2004.

[5] B. Sarwar et al, "Item-based collaborative filtering recommendation algorithms," in *Proceedings of the 10th international conference on World Wide Web*, New York, 2001, pp. 285-295.

[6] S. Owen et al, "Meet Apache Mahout," in Mahout in Action. New York: Manning, 2014, ch. 1, sec. 1-2, pp. 1-3.

[7] The Apache Foundation. (2014, April 19). ItemAverageRecommender.java [Online]. Available: https://github.com/apache/mahout/blob/391cd431dc6b0f2ff1bcdae9f5420c710716b2d4/mrlegacy/src/main/java/org/apache/mahout/cf/taste/impl/recommender/ItemAverageRecommender.java

[8] The Apache Foundation. (2014, April 19). GenericUserBasedRecommender.java [Online]. Available: https://github.com/apache/mahout/blob/trunk/mrlegacy/src/main/java/org/apache/mahout/cf/taste/impl/recommender/GenericUserBasedRecommender.java

[9] The Apache Foundation. (2014, April 19). GenericItemBasedRecommender.java [Online]. Available: https://github.com/apache/mahout/blob/trunk/mrlegacy/src/main/java/org/apache/mahout/cf/taste/impl/recommender/GenericItemBasedRecommender.java

[10] S. Owen et al, "Making Recommendations," in Mahout in Action. New York: Manning, 2014, ch. 4, sec. 1-2, pp. 43-45.

[11] S. Owen et al, "Making Recommendations," in Mahout in Action. New York: Manning, 2014, ch. 4, sec. 4, pp. 56-59.

[12] S. Owen et al, "Making Recommendations," in Mahout in Action. New York: Manning, 2014, ch. 4, sec. 3, pp. 48.

[13] S. Owen et al, "Making Recommendations," in Mahout in Action. New York: Manning, 2014, ch. 4, sec. 3, pp. 50.

[14] T. Dunning. (2007, March 21). Surprise and Coincidence [Online]. Available: http://tdunning.blogspot.com/2008/03/surprise-and-coincidence.html

[15] T. Dunning, "Accurate methods for the statistics of surprise and coincidence," *Comput. Linguist*, vol. 19, no. 1, pp. 61-74, Mar, 2003.

[16] S. Owen et al, "Making Recommendations," in Mahout in Action. New York: Manning, 2014, ch. 4, sec. 4, pp. 55.

[17] G. Lens. (2014). MovieLens [Online]. Available: http://grouplens.org/datasets/movielens

[18] C. Aggarwal et al, "Horting hatches an egg: A new graph-theoretic approach to collaborative filtering," in *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, New York, NY, 1999, pp. 201-212.

[19] J. Leskovec et al, "Recommender Systems," in Mining of Massive Datasets, 2nd ed. New York: Cambridge University Press, 2011, ch. 9, sec. 9.4.2, pp. 327.

[20] N. Good et al, "Combining collaborative filtering with personal agents for better recommendations" in *Proceedings of the sixteenth national conference on Artificial intelligence and the eleventh Innovative applications of artificial intelligence conference innovative applications of artificial intelligence (AAAI '99/IAAI '99)*, Meno Park, CA, 1999, pp.439-446.

[21] Netflix. (2014). Netflix Prize: View Leaderboard [Online]. Available: http://www.netflixprize.com/leaderboard

[22] X. Lam et al., "Addressing cold-start problem in recommendation systems," in *Proceedings of the 2nd international conference on Ubiquitous information management and communication,* New York, NY, 2008, pp. 208-211.

[23] M. Papazoglou, "Service-oriented computing: Concepts, characteristics and directions," in *Web Information Systems Engineering*, Rome, Italy, 2003, pp. 3-12.

[24] J. Rao and X. Su, "A survey of automated web service composition methods," in *Proceedings of the First international conference on Semantic Web Services and Web Process Composition*, San Diego, CA, 2004, pp. 43-54.

[25] R. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. Dissertation, Inform. and Comput. Sci., Univ. of California, Irvine, CA, 2000.

[26] S. Allamaraju, "Appendix Overview of REST," in RESTful Web Services Cookbook, Sebastopol, CA: O'Reilly Media / Yahoo Press, 2010, ch. B, pp. 261-263.

[27] The Apache Software Foundation. (2014, April 10). Welcome to Apache™ Hadoop®! [Online]. Available: http://www.hadoop.apache.org/

[28] The Apache Software Foundation. (2014, April 25). What is Apache Mahout? [Online]. Available: https://mahout.apache.org/

[29] The Apache Software Foundation. (2014, April 9). Spark: Lightning Fast Cluster Computing
[Online]. Available: http://spark.apache.org/

[30] C. Hale and Y. Inc (2014). Metrics: Mind the Gap [Online]. Available: http://metrics.codahale.com

[31] QOS.ch (2014). Simple Logging Facade for Java (SLF4J) [Online]. Available: http://www.slf4j.org/

[32] The Apache Software Foundation (2014). Apache Log4j 2 [Online]. Available: http://logging.apache.org/log4j/2.x/

[33] Project Grizzly (2014, January 24). Project Grizzly: NIO Event Development Simplified [Online]. Available: https://grizzly.java.net/

[34] Oracle (2014). What is MySQL? [Online]. Available: http://dev.mysql.com/doc/refman/4.1/en/what-is-mysql.html