Tallinn University of Technology
Faculty of Information Technology
Department of Computer Science
Chair of Network Software

Juri Kazjulja 120050IAPM

# Recommending board games to groups – prototype and evaluation tool

Master's degree thesis

Supervisor: Ago Luberg
PhD
Assistant

Tallinn
2016

Tallinna Tehnikaülikool
Infotehnoloogia teaduskond
Arvutiteaduse instituut
Võrgutarkvara õppetool

Juri Kazjulja 120050IAPM

# Lauamängu soovitamine gruppidele – prototüüp ja hindamisriist

Magistritöö

Juhendaja: Ago Luberg
PhD
Assistent

Tallinn
2016

# Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

09.05.2016______________ Juri Kazjulja________________

(kuupäev) (allkiri)

# Annotatsioon

Soovitussüsteemid on kaua olnud populaarne teema tarkvaraarenduses. Ettevõtted nagu Amazon, Google, Netflix, Last.fm jmt kasutavad neid süsteeme, et soovitada suurt hulka asju oma kasutajatele.

Teadustöö peamine fookus on olnud süsteemidel, mis soovitavad objekte ühele kasutajale. Grupipõhine soovitus on suhteliselt uus teema ning selliseid töötavaid süsteeme on vähe kasutuses. Selles töös luuakse grupisoovitust pakkuv prototüüp, mida inimesed saavad potentsiaalselt kasutada.

Lisaks luuakse töö käigus tööriist grupisoovituste täpsuse hindamiseks. Nimetatud tööriista saab kasutada erinevate grupipõhiste soovitustehnikate võrdlemiseks.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 54 leheküljel, 4 peatükki ja 8 joonist.

# Abstract

Recommendation systems have been a hot topic in software development for a long time now. Companies like Amazon, Google, Netflix, Last.fm and others use such systems to recommend a large range of items for their users.

However, up until recently all the research on the topic have been centered on designing systems that recommend items to a single user. Research on group-centered recommendation systems is still relatively new and no widely known examples of such systems available for public use exist. This work attempts to change that by creating a prototype of a group-centered recommendation system that would potentially be usable by real people in real world.

In addition, this work provides a useful tool for measuring the precision of recommendations given to a group to compare different recommendation techniques for people that might want to continue work in the field of group recommendations.

This thesis is written in English and is 54 pages long, including 4 chapters and 8 figures.

# Common abbreviations used

- BGG – BoardGameGeek.com website
- RMSE – Root Mean Square Error
- XML – eXtensible Markup Language
- API – Application Program Interface
- POJO – Plain Old Java Object

# List of figures

Figure 1: Aggregating individual recommendations....12
Figure 2: Aggregating user functions in a virtual profile....13
Figure 3: Entity diagram....20
Figure 4: System layout....21
Figure 5: Alternate system layout....23
Figure 6: User name input screen....26
Figure 7: Recommendation list screen....26
Figure 8: Basic flow of the evaluation process....31

# Table of Contents


1 Introduction....10
1.1 Background....10
1.1.1 Neighborhood models....10
1.1.2 Latent factor models....11
1.1.3 Related work....11
1.2 Problem statement....12
1.2.1 Recommending items to users that are united in groups....12
1.2.2 Recommending new items that current user(s) haven't tried....13
1.2.3 Recommending board games....13
1.2.4 Usable in real world....13
1.2.5 Privacy concerns....14
1.2.6 Evaluating recommendation precision....14
1.3 Research questions....15
2 Systems' design....15
2.1 BGG Parser....16
2.1.1 BGG XML API description....16
2.1.2 Getting all valid game ids....16
2.1.3 Parsing games....17
2.1.3.1 Parsing recommended age and player number....17
2.1.3.2 Parsing link tags....18
2.1.4 Parsing ratings....19
2.1.5 Parsing users....19
2.2 Sibyl....19
2.2.1 Database entities....20
2.2.2 System design....21
2.2.2.1 Database interaction....22
2.2.2.2 Aggregation module....22
2.2.2.3 Recommender module....24
2.2.2.4 User interface....26
2.2.3 Running Sibyl....27
2.3 TestSuit....27
2.4 Source code and licensing....28
3. Methodology and experiment design....28
3.1 Evaluating recommenders with Mahout....28
3.1.1 Evaluating predicted rating accuracy....28
3.1.2 Evaluating precision and recall....30
3.2 Experiment design....32
3.2.1 Evaluating predicted rating accuracy....32
3.2.2 Evaluating precision and recall....34
3.2.3 The experiment setup....35
3.2.4 Configurations being tested....36
3.2.4.1 The random recommender....36
3.2.4.2 User neighborhood based with average aggregator....36
3.2.4.3 User neighborhood based with least misery aggregator....37
3.2.4.4 Group sizes....37
3.3 Experiment 1: the random recommender....39

3.4 Experiment 2: user-based recommender with average aggregator........................................41
3.5 Experiment 3: user-based recommender with least misery aggregator..................................43
3.6 Discussion........................................................................................................................44
4. Conclusions and future work..........................................................................................46
4.1 Future work.......................................................................................................................47
Appendix: TestSuit log samples...........................................................................................49

# 1 Introduction

Rapid development and accessibility of the Internet have given us unprecedented availability of information. Today, almost any noteworthy piece of content preserved by humanity in any shape or form is accessible to anyone.

In such a situation, means to filter the content and find the most relevant pieces have become a necessity. Without filtering, people would quickly become overwhelmed by sheer amount of information available to them. One such method, that has gained popularity and is widely used today is collaborative filtering.

The method is based on a simple idea that if two people tended to enjoy similar things in the past, they are very likely to enjoy similar things in the future. Since Tapestry[1], the first system to implement this method, was created, collaborative filtering has gained much attention. Most noteworthy, the Netflix Prize, that had attracted interest of over 20 thousand teams from all over the world[2], has ultimately been won by a collaborative filtering algorithm.

Such interest have led to creation of very sophisticated and precise recommendation engines. For a long time, however, they all had a very serious limitation: they could only produce recommendations for a single user. This means that these systems become significantly less useful if a user wants to choose a piece of content to enjoy with his peers (eg., watch a movie with a group of friends). It was not until late 1990s - early 2000s that the first attempts to tackle the problem of group recommendations were performed[3][4][5].

This thesis aims to apply previously designed techniques for group recommendations on an inherently group-centered domain of board games. As well as offer a practical solution for evaluating the precision of group recommendations.

## 1.1 Background

Academic work in the field of collaborative filtering first began with Tapestry[1], a system that was intended to help filter out “interesting” content from e-mail subscription lists via reviews and annotations. It was a very basic and naive implementation of the collaborative filtering principle, but in the following years interest to the topic grew resulting in more advanced and sophisticated techniques being developed and implemented.

Currently, a wide variety of techniques for collaborative filtering are being used to recommend different types of items. Be it music (Last.fm, Pandora), movies (Netflix, Imdb) or shopping items (Amazon), collaborative filtering helps users easily find items relevant to their tastes.

Most modern collaborative filtering systems use either a neighborhood model or a latent factor model.

### 1.1.1 Neighborhood models

Neighborhood methods focus on finding similar users (user-based neighborhood models) or similar items (item-based neighborhood models) and forming predictions based on this information. User-based neighborhood method is the oldest proposed method for collaborative filtering that is still being used today. It has been proposed by developers of GroupLens recommendation system to filter news articles and movies[6]. This method calculates which users' ratings are most similar to the current user and offer him items that similar users have rated highly.

Item-based neighborhood method is a more recent development on the idea of neighborhood

methods and is focused on finding items that tend to receive similar ratings from same users and use this information to make suggestions. For example, if users that like movie A tend to also like movie B, then it would make sense to suggest movie B to a user that has just rated movie A higly. Item-based methods tend to be preferred as they provide comparable precision to user-based methods while offering a significant computational performance boost[7].

Both neighborhood models rely on similarity functions to calculate the similarity between users or items. Such functions might include, for instance

- Pearson correlation – a number between 1 and -1 that expresses how likely two series of numbers are to move together proportionally
- Euclidian distance, which treats users or items as points in multidimensional space where ratings are coordinates
- Spearman correlation – a variant of Pearson correlation where ratings are first converted in a way where lowest rating of any given user or for any given item is overwritten as 1, second-to-last as 2 etc.[8]

### 1.1.2 Latent factor models

Latent factor models operate under the assumption that items may share some innate characteristics and people may like or dislike them based on which characteristics (factors) are present in a given item and how prominent they are. For example, if a person enjoys comedy movies, has rated several comedies highly and does not seem to enjoy movies that are not comedies, it would seem reasonable to suggest to him other movies that share the comedy genre.

The most popular method using latent factor model is matrix factorization. It first gained large public attention when a research paper using this method has won the Netflix prize. Matrix factorization presents the user-item rating matrix as a product of two matrices where first one represents relations between items and latent factors (i.e how prominent a factor is in an item) and the second one – how much a given user is interested in a factor. The system then attempts to generate such virtual matrices so that the values in the resulting matrix is as close as possible to known user ratings[9].

The beauty of matrix factorization lies in the fact that we don't even need to know what constitutes a latent factor. Be it genre, director of the movie, theme, all we have to do is pick a number of factors and the system will find the relationships on its own.

### 1.1.3 Related work

While there are no group recommendation services available for public use and there were also no attempts, to my knowledge, to create such a service for board games, there are research papers that describe group recommendation services for other domains.

Covered domains include radio stations (MusicFX[4]), movies (Polylens[3]), places to travel (Intrigue[5], CATs[10]) and TV shows (Yu’s TV Recommender[11]). Unfortunately, as was already mentioned, none of these systems are available for public use despite being in differing stages of completion and with Yu’s Recommender even demonstrating prototype screenshots. If one were to google the names of these systems, it is evident that their names are used by other enterprises and only info retrieved that is relevant to the systems are the research papers where they were described.

## 1.2 Problem statement

In this thesis, I had two goals. First one is a proof-of-concept recommendation service that would fulfill the following requirements:

- It should be able to recommend items to users that are united in groups to enjoy together
- It should be able to recommend new items that current user(s) haven't yet tried
- It should recommend board games
- It should be usable in the real world
- It should provide a degree of privacy for users desiring it

In addition to that I aim to create a practical solution to evaluate the average precision of recommendations given to a group.

What follows is a brief overview of already existing solutions that fulfill requirements similar to those previously stated.

### 1.2.1 Recommending items to users that are united in groups

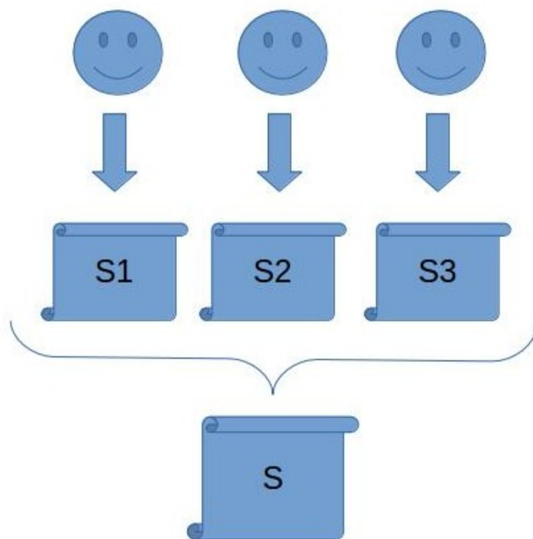



Figure 1: Aggregating individual recommendations

As previously stated, group recommendations is a relatively new field of studies and a definitive solution to the problem of providing suggestions to groups of users is yet to be provided. However, if we look at a survey on existing attempts to tackle the problem, candidate solutions seem to follow two very similar trends.

First one, illustrated by figure 1, consists of forming lists of suggestions (S1 – S3) which are then aggregated into a final list S that is considered as final recommendation list for the given group. Alternatively, a method illustrated by figure 2, first aggregates ratings of users, thus forming a “virtual profile”. Recommendations (list S) are then formed for this virtual profile using traditional collaborative filtering methods.[12][13]

Most existing solutions tend to add some variations to this formula. [14], for example is giving more weight to scores of specific users in a group (such as children or elderly) and [10] incorporates an element of user feedback in the process of forming the recommendations.

The scope of this thesis is applying the group recommendation techniques to a previously unexplored domain, so for now I have decided to implement a basic bare-bones solution and leave more advanced techniques and variations as future work.

### 1.2.2 Recommending new items that current user(s) haven't tried

As stated above, when generating recommendations for user groups, single user recommendations are used as a “backbone”. As such, using state of the art suggestion techniques for one user would definitely benefit the work greatly.

Thankfully, single user recommendations is a well researched topic and several collaborative filtering libraries for a multitude of languages have been developed. One of the most popular and relevant such libraries for Java is Apache Mahout[15], so it is exactly what I decided to use in this work.

### 1.2.3 Recommending board games

Strangely enough, while doing background research on the topic, none of the studies I found were concerned with recommending boardgames to groups of players. There has been some work done considering single user recommendations for boardgames[16], but topic of group recommendations never got explored.

The only service that I was able to find, that does anything remotely similar to what this thesis is trying to accomplish is gameshelf.se. They, however, focus on a different aspect of suggestions: recommending games to play in a group from combined collections of the participants. Only in April 2015 have they released a feature that suggests new games to buy. It is unclear which method they used, but they don't seem to mention collaborative filtering in any form on their website or trello board[17]. It would still be interesting to compare effectiveness of suggestion mechanisms as future work.

Unfortunately, my attempts to reach out to the creators of the service for possible collaboration were not answered.

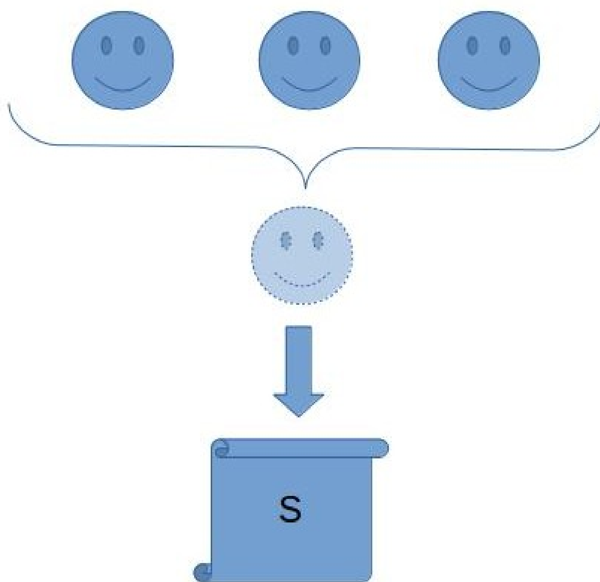



*Figure 2: Aggregating user functions in a virtual profile*

To test my solution out, I have decided to take an approach similar to that of authors of [16] and copy game, ratings and user data from BoardGameGeek[18] website, as it is the biggest and most popular board game database.

### 1.2.4 Usable in real world

As already stated in subsection 1.1.3 (page 11), current group-centered recommendation system implementations tend to be unavailable for public use. The only way we do know of their existence is through the research papers describing them. In fact, I was unable to find a single example of collaborative filtering powered group-centered recommendation system that would be available for public use.

While this in no way makes these authors’ contribution to the field any less valuable, I still believe that it is a shame that in our highly social and connected world recommendation systems still remain almost completely individualistic and don’t provide as much value to groups as they could have using already developed techniques and solutions.

This is why I have, right from the start, built the system keeping in mind the fact that it is aimed at

production environment and made sure that it would be as ready for wide use as early as possible. To achieve this, I have attempted to make the system so that it would be usable to, at least, people who already have BoardGameGeek accounts even if it would be hosted as is, in early prototype form.

### 1.2.5 Privacy concerns

User privacy is one of the important topics that should be considered when designing recommendation systems. Users might not want other people in the group to know that they like or dislike certain items to avoid embarrassment and it should not be too easy to deduce such info from the suggestions the system provides.

That being said, BoardGameGeek, strangely, does not provide any kind of privacy settings for its users: all the info that a user has entered into the service is freely available to everyone. So, because prototype system is going to use BGG data, putting in place privacy settings just yet does not really make a lot of sense.

This is why I have decided to leave this particular requirement out of the scope of this work.

### 1.2.6 Evaluating recommendation precision

Quite obviously, to answer one of the research questions (section 1.3, page 15) we need some way to know which recommendation method gives better results. Some studies, eg. [12] use user surveys to determine which system provided results that were perceived as most interesting. As I don't have resources to conduct such surveys, I had to find an alternative way to evaluate recommendation methods.

The method I chose is similar to the one Mahout is using to evaluate their recommenders. In fact, I ended up heavily basing my solution on Mahout code. The method consists of splitting the rating information in some proportion and attempting to predict one part of the ratings using the other. Naturally, the smaller the difference, the better our recommender is.

We can not, however, apply the method naively, as we do not have have any “group ratings” to compare predicted value to. So we have to somehow calculate the difference using real ratings of group members. I have decided to base this calculation on the method used in [13] to evaluate the effectiveness of group recommendation method proposed in that work.

Method consists of calculating RMSE

$$RMSE=\sqrt{\frac{\sum_{i=1}^{n}(p_i-a_i)^2}{n}}$$

where $p_i$ is predicted rating for a given item, $a_i$ is the actual rating group members have given the item and $n$ is the size of the group.

The smaller RMSE we get, the more precise our prediction was.

Author of the aforementioned work does not go into detail on practical aspects of doing this evaluation. Whereas I aim to provide a free and open solution for automatic evaluation of group recommendation systems that could be easily used out of the box provided that the system being evaluated can be tweaked to extend 2 simple and minimalistic interfaces I have introduced. Which should not be a problem if said system is using Mahout as a backbone.

In addition to calculating RMSE I have also decided to implement the calculation of other metrics

that is also present in Mahout's evaluation toolbox: calculation of precision and recall. In statistical analysis, precision refers to the ratio between the amount of relevant selected elements and total amount of items selected, whereas recall is the ratio between selected relevant items and total relevant items present.

Precision and recall, while relevant on their own, allow to calculate one of the most popular measures of test effectiveness that is called “F-score”, that is calculated as follows:

$$Fscore=\frac{2\cdot P\cdot R}{P+R}$$

where P stands for precision and R stands for recall.

One might even argue that in the situation where precise score to compare the estimated one to is unknowable, F-score might offer a more reliable measure for system effectiveness.

## 1.3 Research questions

***RQ1: Which aggregation strategy will perform best for the domain of board games?***

In attempts to tackle group recommendation problem several aggregation strategies were proposed. As far as I am aware, no consensus exists on which strategy produces more accurate results. The 2 research papers I have found that have attempted to compare performance of different strategies[12][13] have reached different conclusions. Additionally, none of papers dealt with boardgames domain.

Taken all that into account, it seems wise to conduct my own experiments and find the strategy that will produce more accurate results with my particular dataset.

***RQ2: How can we automate evaluation of group-centered recommendation systems?***

As there is no definitive solution to the problem of group-centered recommendations, it is important that a person working on said problem would be able to reliably and automatically test different candidate solutions.

I haven't been able to locate any openly available tool for that, so I believe that creating such tool would bring great value to anyone who is working in the domain of group-centered recommendation systems.

# 2 Systems' design

While attempting to fulfill all the objectives I have set for my self in this thesis, I ended up creating 3 separate applications.

- BGG Parser – a tool that I used to get initial evaluation data. It goes through BoardGameGeek catalog of boardgames and uses the BGG XML api to get all the relevant information and save it in the local database.
- Sibyl – the recommendation system itself. It operates as a web-service that, given several BoardGameGeek user names can calculate a list of up to 20 (configurable) games that provided users are all likely to enjoy based on what information is already present in the database. The name “Sibyl” has been chosen as a homage to a Japanese animated TV-show “Psycho-Pass”.

- TestSuit – the recommender evaluation tool. It creates random groups of users, attempts to predict the rating of a random game for the group and calculates RMSE using actual ratings of group members.

What follows below is a more in-depth explanations of said applications' design.

## 2.1 BGG Parser

BGG Parser is a tool that I have created to call BoardGameGeek XML api and populate my database with information on users and games that they have rated.

### 2.1.1 BGG XML API description

BoardGameGeek provides an API – located at http://www.boardgamegeek.com/xmlapi2/ – that allows to fetch data concerning various items present in BoardGameGeek database. The general pattern that the API accepts is http://www.boardgamegeek.com/xmlapi2/<type>?<parameters>.

Types that are relevant and were used in this work are:

- `thing` – "any physical tangible product"[19] including boardgames, boardgame expansions, accessories etc.
- `user` – for information on users and their friendships. Friendship info was originally planned to be used in creating test data, however I ended up using a different method. As such, the friendship data, though gathered, will not be used.

As for parameters, with "things" I used:

- `id` – self-explanatory. Id of the requested "thing".
- `ratingcomments` – parameter that, when set to 1 tells the api that we desire to get user ratings and review scores for the "thing" we request.
- `pagesize` – when requesting ratings, the api does not return all the ratings for the item, but presents a "page" with some number of ratings. This parameter sets the amount of ratings per page. 100 is the maximum, so I used that.

For users I used the following parameter:

- `buddies` – displays which users are friends of requested user. As with ratings for things, results are paged. Api does not, however allow to set a page size for users.

For both users and "things" I have used the `page` parameter, which specifies, which page are we requesting.

While providing almost all the data I needed for creating a proof-of-concept application and testing, there is one thing the api did not offer. And that is a list of all valid game ids. So I had to find a different way to get them.

### 2.1.2 Getting all valid game ids

After trying different tricks to achieve the goal of getting all the game ids, I have stumbled upon a collection of sitemap files at https://boardgamegeek.com/sitemap_geekitems_boardgame_page_X, where X is a page number from 0 to, at the moment of writing of this paper, 9. These files, intended to be given to search engines for indexing, contain links to every boardgame page on the site. And

those links contain the ids of the games, so getting all of the valid ids was just a question of parsing them out of the sitemap files.

Using dom4j I have been able to get games urls located at /urlset/url/loc in the sitemap files. And since all of the urls follow the pattern http://boardgamegeek.com/boardgame/<gameId>/<gameName>, getting the id was as easy as making a substring from character number 35 (length of “http://boardgamegeek.com/boardgame/” part of the url) to `lastIndexOf(“/”)`.

**2.1.3 Parsing games**

Game is presented by the api as an XML document where different properties are stored as values of elements. Most of the properties are quite easy to get: it's just a matter of fetching value using xpath that looks like //propertyName/@value while performing necessary null-checks.

Another thing to consider is that not all would-be numerical values returned are strictly numerical. For example, if a game supports a huge number of players, the maxplayers property might contain the value “10+”, which means “more then ten”. Using parseInt on such a string would lead to format exceptions and the plus sign should be trimmed.

In such a manner I was able to get values for game name, publishing date, minimum amount of players, maximum amount of players, average playing time (in minutes) and minimum age. This, however, was not enough for the recommended player age and recommended player number as well as several metadata elements that are provided via “link” tags.

**2.1.3.1 Parsing recommended age and player number**

To understand why these two attributes are different, let us look closely at how they are represented by using an example from an XML that represents the game “Terra Mystica”. Here is the listing for recommended player number:

```
<poll name="suggested_numplayers" title="User Suggested Number of Players" totalvotes="482">
  <results numplayers="1">
    <result value="Best" numvotes="0"/>
    <result value="Recommended" numvotes="10"/>
    <result value="Not Recommended" numvotes="248"/>
  </results>
  <results numplayers="2">
    <result value="Best" numvotes="14"/>
    <result value="Recommended" numvotes="181"/>
    <result value="Not Recommended" numvotes="164"/>
  </results>
  ...
  <results numplayers="5+">
    <result value="Best" numvotes="6"/>
```

```
    <result value="Recommended" numvotes="9"/>
    <result value="Not Recommended" numvotes="170"/>
  </results>
</poll>
```

And here is the listing for recommended player age:

```
<poll name="suggested_playerage" title="User Suggested Player Age" totalvotes="120">
  <results>
    <result value="2" numvotes="0"/>
    ...
    <result value="10" numvotes="12"/>
    <result value="12" numvotes="47"/>
    ...
    <result value="21 and up" numvotes="0"/>
  </results>
</poll>
```

As is evident from the listings, the values of these attributes are stored in a form of a result of a poll of BoardGameGeek users' opinions on the matter. While this information might be useful in some contexts, I have decided against including its use, or, in fact, collection, in the scope of this work. As such, I only really needed one value per attribute, so, to get values for these particular attributes, I ended up iterating through the result tags and saving values that had the most votes. In case of ties I kept the value that came up first.

In addition to that, for recommended player number value I had to iterate through the results tags instead of result and check the numvotes attribute for the tags with value “Best”.

#### 2.1.3.2 Parsing `link` tags

Link tags are a way to express the relationships that games might have with one another. For example, if two games have the same designer, this will be expressed by them both having a `link` tag of the same type, id and value. These relationships are not used in the work yet, but their incorporating into recommendation-making process is most certainly a promising direction for future development of the Sibyl application, so I have decided to save them too.

The possible types link tags might have include the following:

- `bardgamecategory` – for information on broad categories the game falls into. For example, if it is fantasy themed, it will be in the “Fantasy” category. Likewise, if it is a card game, it will be in the “Card Game” category.
- `boardgamemechanic` – different gaming mechanics that are present in the game. For example, this category could tell us if the game involves dice.
- `bordgameartist` – names of people who did the artwork for the game

- `boardgamedesigner` – names of the people who designed the game
- `boardgamepublisher` – names of companies that published the game
- `boardgamefamily` – "family" of game might indicate that it is thematically related with other games and/or are set in a same fictional universe

For the Sibyl to be able to easily recognize that several items are connected by any of the links, I had to create them as a separate entity in the database and connect this entity with games via a reference table.

### 2.1.4 Parsing ratings

All ratings are treated as a part of a comment by BoardGameGeek, even if there is no review provided with the rating. So I had to iterate through the `comment` tags to get the information I needed.

What made that tricky is the concept of pagination that the api has. So I had to keep track of how many total ratings an item has and how many pages, knowing the amount of items per page, would that leave for me to parse.

I also have chosen to insert the user parsing part of the information gathering process to be triggered when I encounter a rating by a user that we haven't encountered before. Which brings us to the next subsection.

### 2.1.5 Parsing users

This part has been quite straightforward with only information being saved being user name, country of residence and friendships. The only non-trivial part were the friendships. In addition to being paginated, and, as such, being a subject to a process described above in the Parsing ratings part, I also had to deal with the issue of every friendship appearing in the data twice due to inevitable situation of parsing both parties' information.

I mitigated this problem by only saving friendships to users that I already had parsed, so a friendship would be processed only once.

As already mentioned above, friendship information has not been used in the work just yet, as I have chosen to use simpler methods. However, using location data and friendship status could serve as a more realistic way to create testing user groups and, as such, is an interesting and promising direction for future work.

## 2.2 Sibyl

The proof-of-concept group-centered recommendation application that I have originally set out to create.

It takes BoardGameGeek user names as input and, treating presented users as a group seeking recommendation, attempts to suggest them new games to try using information that has been parsed by BGGParser and saved into Sibyl's own database.

### 2.2.1 Database entities

All of the data parsed by BGG Parser has been used to fill the Sibyl database, which can be described by Figure 3.

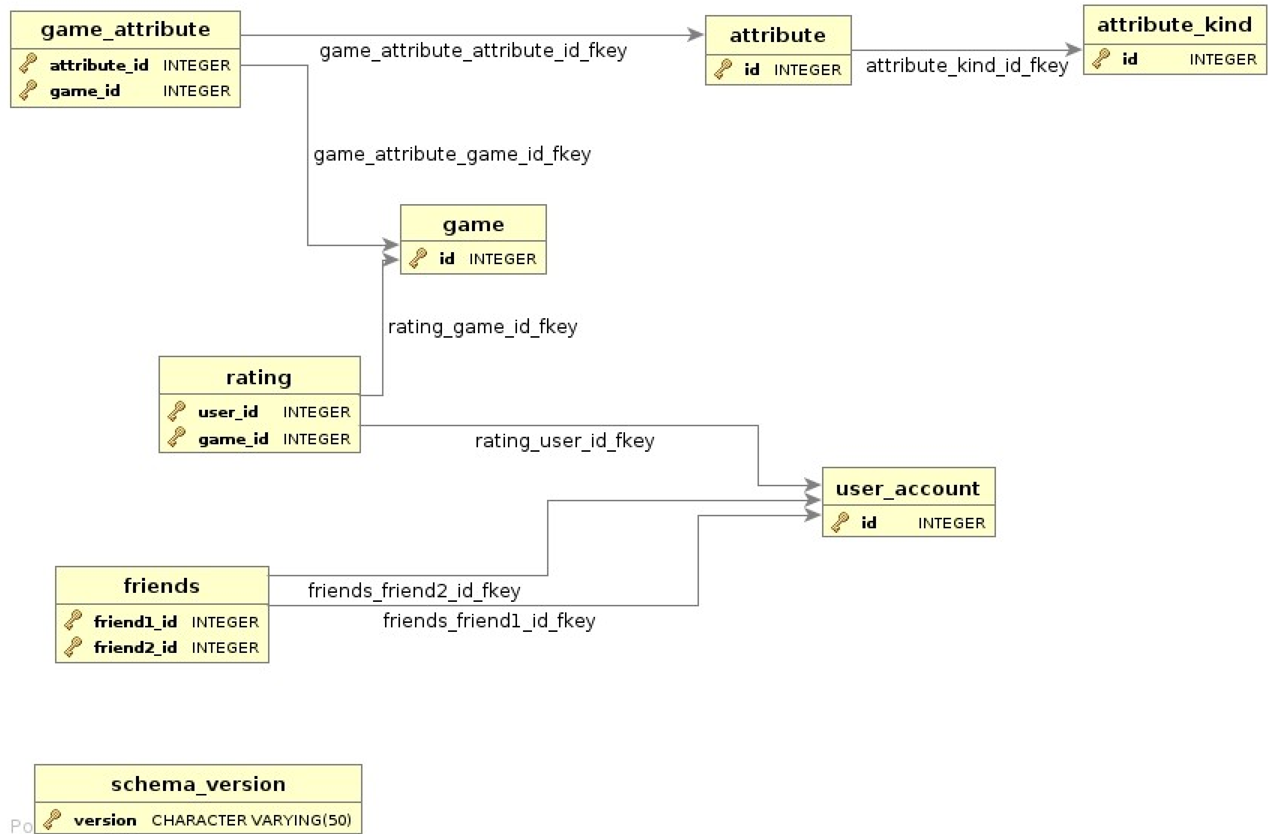



*Figure 3: Entity diagram*

Game, rating and user_account seem to be self-explanatory. Game table contains information on games, user_account describes users and rating contains information on how a user rated an item.

Friends is a table that represents which users are friends. Attribute contains all the attribute values from the link tags in game xml document and links to attribute_kind table for information about the kind of an attribute it is (eg. name of the designer, game mechanic, etc). Game_attribute table maps games and their respective attributes.

Schema_version is a servicing entity that was generated by flayway[20] database migration framework tool that I have used for database version control.

### 2.2.2 System design

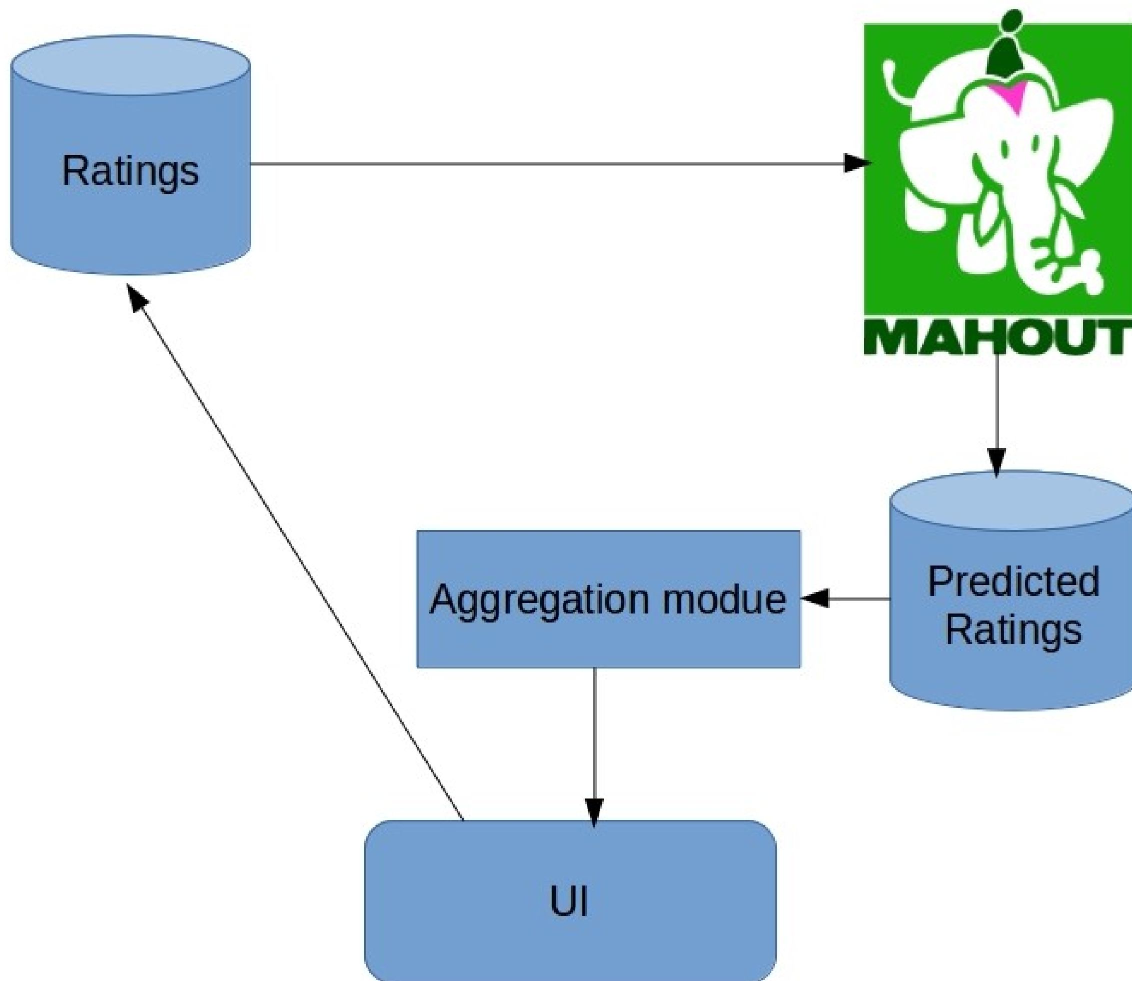



*Figure 4: System layout*

Figure 4 illustrates interactions between different system components.

Ratings are first processed by mahout and recommendations for every user are provided. Recommendations are then aggregated and presented to logged-in users via the UI. In following versions it is assumed that users will be able to rate the items and therefore fill the ratings database, but in the current proof-of-concept implementation this part is not implemented.

An attentive reader might recognize that this particular design follows a group-centered recommendation system building principle described by Figure 1 back in the first chapter, ie building a list of recommendations for single users first and then aggregating them.

The reasoning behind choosing this particular model and not the “virtual” user alternative comes from a simple fact that I have used Mahout as single-user recommendation building foundation that is needed for both models. And it appears to me that in the particular case of Mahout it is somewhat easier to perform manipulations with recommendations it provides then presenting it a new user in a way that does not involve saving him to the database.

What follows is a more in-depth exploration of all the system's parts.

#### 2.2.2.1 Database interaction

For database interaction I have used the jOOQ framework. This framework generates java code from database metadata and allows to write queries to the database using pure java and getting query results as POJOs. For instance, to get a game record from the database by id I would use

```
create.select()
	.from(GAME)
	.where(GAME.ID.equal(itemID.intValue()))
	.fetchOne()
	.into(Game.class)
```

This statement returns a Game object that describes a game which has id itemID in the database.

One of the strengths of the jOOQ framework and one of the reasons I have ended up using it is because, while doing all the queries using native java code, queries themselves write and read much like SQL statements and are, as such, easily readable and understandable.

All the database calls needed for Sibyl to properly work are contained in `DbConnector` object. These methods are

- `getGameById(Long itemID)` returns a `Game` object (automatically generated by jOOQ using Game table from the database) that represents a database record from the table game with provided ID. This is needed for displaying game names when presenting recommendations
- `getUserByName(String name)` similarly to the method above, returns a `UserAccount` object (also auto generated by jOOQ) that represents a user with a given name. This is used to get user IDs when requesting recommendations.

#### 2.2.2.2 Aggregation module

Picking the right aggregation function can make a difference between an ok group-centered recommender system and a great group-centered recommender system. As such, it is imperative that aggregation modules would be easily swappable.

To achieve this, I have made it so, that the function of aggregation module could be fulfilled by a single java class, instance of which would then be provided to an object of a Recommender class (described below) on creation. Only prerequisite that a class needs to fulfill to work as an aggregation module is it has to extend the `Aggregator` interface.

`Aggregator` is a simple interface that only has one method signature:

```
float aggregate(List<Float> ratings)
```

as should be evident from this signature is that the job of an Aggregator in a system described above is to take a list of predicted ratings in float format, apply an aggregation function to them and return the result in float format that is then treated as a final predicted rating and is passed on to the UI.

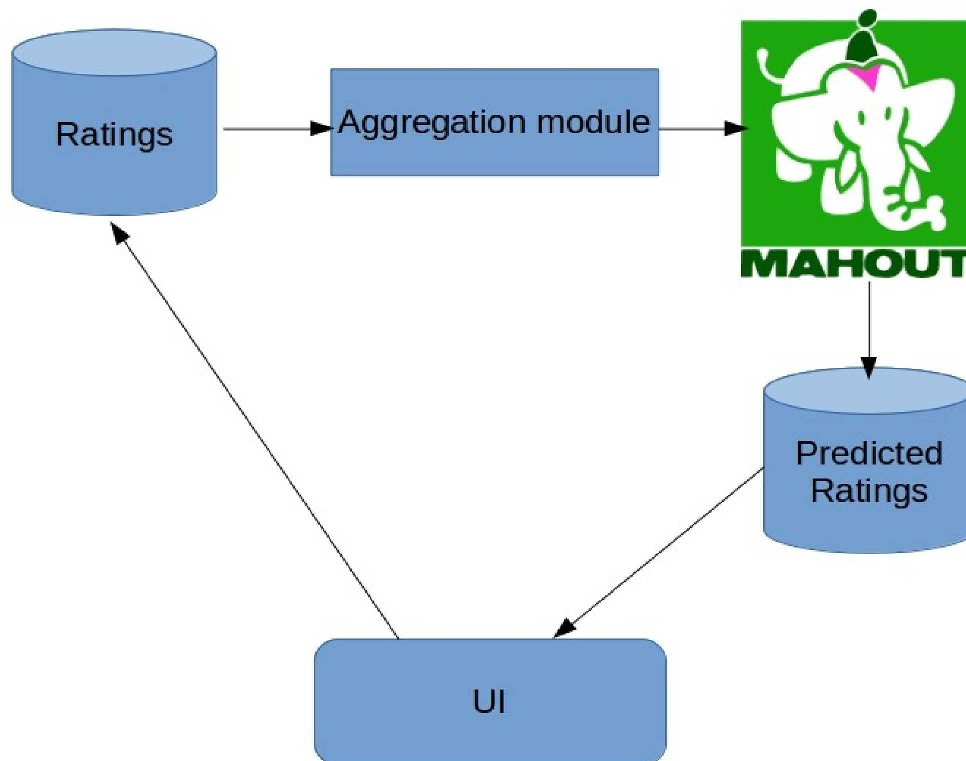



*Figure 5: Alternate system layout*

The advantage of such a design is that the aggregator does not really care if the ratings provided are predicted or actual. So, if we were to try to reimplement the system in a way reflected in Figure 5, as in, implementing a “virtual user” model, we could use the same aggregators we already have without any extra work.

Right now Sibyl uses `AverageAggregator` class as an aggregating module. It is evident from the name that this aggregator returns an arithmetic average of all ratings presented.

One thing to consider when writing aggregators is that not all values in presented list might be strictly numeric. If we provide a list of ratings of different users on one item while creating a virtual profile, some users might have not rated the item being processed. Or, if we try to aggregate recommendations on one item for a group using estimates that Mahout's single-user recommender has provided, there could be NaNs in the list, as sometimes Mahout is unable to predict a rating for a given user on a given item and returns a NaN in such case.

In my implementation of the `AverageAggregator` I have chosen to treat NaNs as if they are not there at all. For example, when presented with a list [2, NaN, 4, 5, NaN], the resulting calculation will be

$$\frac{2+4+5}{3}=3,(6)$$

Weather or not this way to deal with NaNs is the best one is up for debate. Possible alternatives might include treating NaNs as zeros, making the calculation

$$\frac{2+0+4+5+0}{5}=2,2$$

or treating it as an “average-neutral” value. In five point system, for example, such a number would be three resulting in

$$\frac{2+3+4+5+3}{5}=3.4$$

I believe that experimenting with different approaches of dealing with NaNs could be an interesting direction for future work. Indeed, some other method that I did not think of might prove to be the best and most effective one.

#### 2.2.2.3 Recommender module

The recommender module is the most important part of the Sibyl application. This is where most calculations take place and this is the part that provides recommendations for UI to display.

As with aggregators, my aim was to make recommender modules as swappable as possible, as I was intending to try out different recommendation techniques until I find the one that produces the best results. So I started with defining the `GroupRecommender` interface, that is in turn based on Mahout’s `Recommender` interface.

The `Recommender` interface defines methods like

- `recommend(long userID, int howMany)` that provides a list of recommended items for user `userID` that contains, at most, `howMany` items, or less if Mahout is unable to provide this number of items
- `estimatePreference(long userID, long itemID)` that attempts to predict, what rating user `userID` might give to an item `itemID` and returns NaN if Mahout is unable to form a prediction for whatever reason. Recommendations are returned as a list of `RecommendedItem` objects, that contain a pair of values: ID of the item being recommended and the estimated preference value.

I have extended this interface into my own `GroupRecommender` interface that defines the following methods:

- `recommend(Collection<Long> userIDs, int howMany)`
- `estimatePreference(Collection<Long> userIDs, long itemID)`

Similarity to method signatures from the `Recommender` interface are not a coincidence. Indeed, the intended purpose of these methods is to do the exact same thing `Recommender` methods do but for a group of users that is defined by collections of userIDs provided.

Using such simple and unrestrictive interface I have ensured that taking out one recommender and replacing it with another is as easy as changing an aggregator. One might wonder why I have chosen to extend Mahout’s `Recommender` interface instead of defining one from scratch. The reasoning behind this decision was that this enables me to use to use Mahout's recommender evaluators as a basis for my own. Evaluators will be discussed further in part of my work dedicated to the TestSuit.

Next step was to provide and implementation for the interface. As already noted, I have chosen to start with implementing the recommendation list aggregation model. For this I needed an aggregator (described in previous subsection) and a “base” recommender for forming single-user recommendation lists.

Thankfully, Mahout provides a selection of throughly developed and effective single-user

recommenders with most noteworthy being

- `GenericUserBasedRecommender` that serves as an implementation of user-based neighborhood model
- `GenericItemBasedRecommender` that serves as an implementation of item-based neighborhood model
- `SWDRecommender` that serves as an implementation of the latent factor model

All of these recommenders can be further customized by providing different strategies on recommender creation. For example, recommenders implementing neighborhood methods accept a variety of similarity metrics that are also provided by Mahout. Namely, all similarity metrics discussed in subsection 1.1.1 (page 10), as well as several others, are freely available for use. In addition to this user-based recommender accepts different neighborhood type objects that determine how similar users need to be to each other to be considered neighbors (threshold neighborhood) or how many neighbors a user should have (nearest N users neighborhood).

`SWDRecommender` can also be customized via providing factorization techniques that are based on different research papers on topic of matrix factorization.

The implementation of `GroupRecommender` that Sibyl uses right now is called `RecommendationAggregatingRecommender`. On creation it receives an aggregator (any implementation of an `Aggregator` interface) and a base recommender for single-user suggestions (any implementation of Mahout's `Recommender` interface) as constructor arguments.

Currently, as a base recommender I use `GenericUserBasedRecommender` with Euclidian distance similarity metric and nearest N user neighborhood with N = 100. It will most probably turn out, after extensive testing, that this is not the best choice of an algorithm, but it is good enough for proof-of-concept and architecture of the system makes changing the recommender as easy as defining a new one and providing it to `RecommendationAggregatingRecommender` as a constructor argument.

Aggregatin recommender’s `estimatePreference` method works simply enough. It takes the user ID collection provided and item ID to be estimated. It then iterates through the user ID collection, calling underlying base recommender’s `estimatePreference` method for every user ID and the provided item ID. All the results are then saved into a list that is then passed to the aggregator. Finally, the result of aggregation is returned as the estimated preference.

The `recommend` method starts off similar to the `estimatePreference`. It iterates through user ID list, calling the underlying recommender’s `recommend` function and gathering the resulting `RecommendedItem` objects. This, however, is where similarities end.

Because we call `recommend` for every user separately, we end up with list that may not contain the same items. Because of this, I gather all the recommended item lists in one and, using java’s new lambda functionality, collect all the item IDs in the list into a set. Set was chosen because the next step in the process will be iterating through item IDs to determine estimated preference for the whole group and iterating through an item twice is counter-productive.

After item IDs have been gathered, we walk through them determining an estimated preference for every user of the group on this item and feed these preferences to the aggregator. Much like we did in the `estimatePreference` method.

Aggregated preference and item ID are then stored into a new `RecommendedItem` object instance, that is then stored in a list that will form the result of the recommendation process. This

list is then sorted by preference in a descending order and cropped so that it would not be bigger then `howMany` provided on method call.

#### 2.2.2.4 User interface

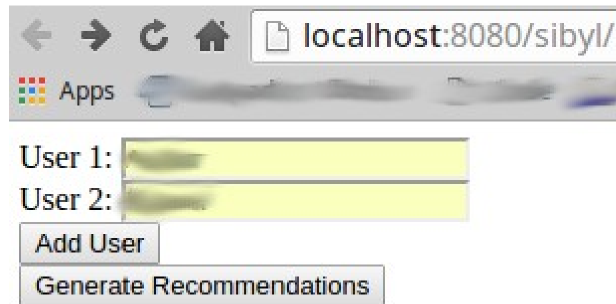



*Figure 6: User name input screen*



| Name | Est. score |
|---|---|
| Advanced Squad Leader | 9.5 |
| Mansions of Madness: Season of the Witch | 10.0 |
| Hive: The Mosquito | 8.5 |
| Neuroshima Hex! | 8.571428 |
| The Downfall of Pompeii | 8.238497 |
| Carcassonne Big Box 4 | 10.0 |
| Robinson Crusoe: Adventures on the Cursed Island | 8.549749 |
| Dune | 9.192307 |
| Carcassonne Big Box 5 | 9.6 |
| Cards Against Humanity: Sixth Expansion | 9.818182 |
| Icehouse | 9.0 |
| Arkham Horror: Innsmouth Horror Expansion | 9.6 |
| Chaos in the Old World: The Horned Rat Expansion | 9.539504 |
| Can't Stop | 8.5 |
| Wir sind das Volk! | 10.0 |
| Jaipur | 8.195746 |
| Combat Commander: Europe | 8.5 |
| Legendary: A Marvel Deck Building Game | 8.859276 |
| Freedom: The Underground Railroad | 10.0 |
| Kolejka | 9.75 |
| The Duke | 9.666667 |
| Glory to Rome | 8.713474 |
| Spartacus: The Serpents and the Wolf Expansion Set | 9.6 |
| Winner's Circle | 9.0 |
| Firefly: The Game ? Big Damn Heroes Promo Cards | 10.0 |
| Mice and Mystics: Heart of Glorm | 9.637334 |
| DC Comics Deck-Building Game: Crisis Expansion Pack 2 | 9.620689 |
| Ace of Aces: Handy Rotary Series | 8.5 |
| Galaxy Trucker: The Big Expansion | 9.396656 |
| Starfarers of Catan | 9.666667 |
| Cosmic Encounter | 8.604166 |
| Age of Empires III: The Age of Discovery | 8.382556 |

*Figure 7: Recommendation list screen*

As the Sibyl application is still in prototype stage, the user interface is quite minimalistic and not very exciting. It consists of just two pages. First page, presented on Figure 6 contains fields to add user names to form a group, a button to add another input field for the next group user and a button to get recommendation list.

Figure 7 illustrates a sample result of requesting recommendations. When this screenshot was taken, sorting the recommendation list by estimated score was not yet implemented, so the recommendations will appear unsorted.

As could be seen, even in it's simple form, the system is in fact already fit for in-house use.

As for what happens "under the hood", most of the work is done by `RecommendationPage` servlet. It contains an instance of aggregating recommender and `DbConnector` (described in subsection 2.2.2.1, page 22).

These instances are created in the servlet's initialization method. It should be noted that this servlet should be set to initialize at application startup. This is required because I have chosen to use caching implementations of user similarity and neighborhood objects. These particular implementations offer a significant performance boost because they load some of the similarity information into memory on startup. The trade-off is that they take significant time to initialize. So if we don't initialize the servlet on startup, first time a user tries to get recommendations, the page won't respond for quite some time.

When presented with a list of user names via request parameter, servlet first asks `DbConnector` to get him user account objects. It then forms the list of user IDs and requests the recommender (described in the previous subsection) to provide a list of recommended items. When given the list, servlet again requests `DbConnector` to provide game objects for item IDs that are in recommended items list.

Finally, it uses the data from the collection of game objects and recommended item objects to form a final recommendation list that it then presents to the user.

### 2.2.3 Running Sibyl

To run the Sibyl application I use the Jetty application server that I start using a Gradle plug-in. Since Sibyl itself is quite simple and bare-bones right now, it makes no sense to use other, more advanced application servers.

## 2.3 TestSuit

There are a lot of variables when constructing a group-centered recommendation engine. Even if we take only one of two models presented in subsection 1.2.1 (page 12), there are a lot of different aggregating functions one might try. In addition to just returning the average value, as Sibyl currently does, there are also functions such as "least misery" – choosing the smallest value out of those presented; or a variant of the two, that is "average without misery" – taking average of the ratings but excluding from recommendation the items whose ratings are below a certain threshold. As well as numerous others.

In addition to that, as noted in the subsection 2.2.2.3 (page 24), Mahout offers a selection of recommenders that are – by the use of different similarity metrics, neighborhood determination techniques and factorization strategies – approaching infinity in numbers.

And, if we take into account that there are two possible applicable models and every each variable

noted above is applicable to both of them, we end up with a myriad different ways to build a system and no way to easily determine which architecture is better for us.

That is why I decided to create TestSuit. The tool to automatically run tests on a recommendation system and measure the precision and relevance of the results it produces.

The architecture is quite simple. It consists of several evaluator classes that are used to produce different kinds of metrics about the system at hand and a runner class that is to be redefined by a person wishing to use it.

It also comes pre-packaged with several useful interfaces, such as `GroupRecommender`, `Aggregator` and `GroupDataProvider` (this one will be described and discussed later) as well as some sample implementations.

I could describe the inner workings of the evaluator here, however as the inner logic is identical to the experiment design that I plan to propose, I believe that it would be more logical to put this explanation to the relevant part of the work.

## 2.4 Source code and licensing

A version of the system's source code used in this thesis is available at https://bitbucket.org/XonX/sibyl-public and is distributed under MIT license.

I have considered choosing a more restrictive license for the web application part of the work, as I plan to make a standalone commercial service out of the project. However, none of the software licenses that I have found quite fit the requirements. Also, after considering the amount of work needed to transfer the prototype that I currently have to a full-fledged product, I came to the conclusion that it is safe to release the code under a permissive license.

Locking the automatic evaluator part before a strict license would defeat the purpose of the work as there is no real value in providing a tool that no one can use.

Sibyl and TestSuit both have copies of required interfaces so anyone willing to use or develop TestSuit does not need to have a copy of Sibyl on their machine.

# 3. Methodology and experiment design

As already briefly noted in subsection 1.2.5 (page 14), the main point of experiments is to calculate RMSE between predicted group rating and actual group member ratings as well as calculating precision and recall that can then be used to calculate F-score.

As I have heavily based my testing framework on Mahout's recommender evaluation functionality, I believe it would be beneficial to start with a brief overview of how Mahout's recommender evaluators work.

## 3.1 Evaluating recommenders with Mahout

### 3.1.1 Evaluating predicted rating accuracy

To evaluate, how precisely a recommender predicts ratings for its users, Mahout uses different implementations of the `RecommenderEvaluator` interface. I have used

`AverageAbsoluteDifferenceRcommenderEvluator` as a base for my evaluator so this is the evaluator I will be mainly describing in this subsection.

Before an evaluator is invoked, an instance of a `RecommenderBuilder` must be created. The purpose of the builder is to delay the recommender creation as it has to be supplied with a data model that is modified and different from the one we have in the beginning.

Getting an evaluation is as easy as calling the `evaluate` method which has the signature

```
evaluate(RecommenderBuilder recommenderBuilder,
         DataModelBuilder dataModelBuilder,
         DataModel dataModel,
         double trainingPercentage,
         double evaluationPercentage)
```

`RecommenderBuilder` has already been discussed above. As for the rest of the arguments

- `DataModelBuilder` – an optional argument that provides an ability to specify a custom data model builder. If this argument is null, a default Mahout builder is used. I decided not to experiment with custom builders just yet, so this argument will not be used
- `DataModel` – Mahout's representation of the data model. This object is used to store and retrieve all the information relevant to the evaluation process. Namely, user IDs, item IDs and the ratings. Mahout provides many different implementations of `DataModel` including one to work with data presented as a comma separated file or as a column in a database
- `trainingPercentage` – the percentage of each user's preferences to produce recommendations. For each user used for evaluation this percentage of ratings will remain as base for predictions
- `evaluationPercentage` – percentage of users to use in evaluation. Ideally, we would like to test the whole user base that we have, but this might take large amounts of time. Using this parameter we can "shrink" our user number to a more manageable size so we don't have to wait for days to check if some small tweak in one parameter of the evaluator made any changes

It is important to note that `trainingPercentage` and `evaluationPercentage` are not strictly connected to each other and, as such, do not have to add up to 1 (100%).

For example, it is completely possible to call the `evaluate` method with `trainingPercentage = 0.8` and `evaluationPercentage = 0.4`, making their sum 1.2. All this means is that we will take 40% of all the users in the system and then we will split away 20% of their ratings and try to predict them using the remaining 80%.

When the `evaluate` method is invoked, evaluator starts walking through all the user IDs and "rolling the dice" for every each one of them using `random.nextDouble()`. If the number produced by the random number generator is lower then the `evaluationPercentage` provided, the user gets chosen as one of the test users and his ratings are then spitted between two sets of preferences: the training preferences and the test preferences.

Splitting is somewhat similar to test user selection. For a given user, evaluator goes through all of his ratings, again, generating random double values but this time, using `trainingPercentage` as a threshold. If a generated value is smaller then `trainingPercentage`, the preference is put in a collection of training preferences and, therefore, will be used later to predict test values. If the randomly generated double value is larger then `trainingPercentage` it is stored in test

preference collection and the recommender being tested will then later try to guess it.

After the splitting is complete, the collection of training preferences is then provided to data model builder (custom if provided, default otherwise) which then generates a new data model to be handed to the recommender using the recommender builder.

The evaluator then takes the freshly created recommender and walks it through the testing preference collection making it attempt to predict every rating spitted away previously. It then takes the absolute difference $|e-a|$ , where e – is the value expected by the recommender and a – is the actual value in the test preference collection.

After all the evaluations are done, the average value of all calculated absolute differences is presented as the result of the evaluation process.

### 3.1.2 Evaluating precision and recall

To calculate precision and recall, Mahout employs an evaluator class named `RecommenderIRStatsEvaluator`.

As with `AverageAbsoluteDifferenceRecommenderEvluator`, to use it, the recommender class being evaluated should be first prepared for actual construction via a `RecommenderBuilder`. This, however, is where similarities end. While still having an `evaluate` method, the actual signature is quite different:

```
evaluate(RecommenderBuilder recommenderBuilder,
         DataModelBuilder dataModelBuilder,
         DataModel dataModel,
         IDRescorer rescorer,
         int at,
         double relevanceThreshold,
         double evaluationPercentage)
```

`RecommenderBuilder`, `DataModelBuilder`, `DataModel` and `evaluationPercentage` should already be familiar from the previous subsection. Indeed, their function is almost identical to those described previously, so I will not repeat their description.

As for the new parameters,

- `rescorer` – rescorers are used in Mahout to further customize the recommendation process. If a rescorer is supplied to a recommender it can be used to filter out some results that the recommender returns or assign new scores to items based on specific cases. I did not use rescorers in this work, so I will not be discussing them in detail.
- `at` – as was explained in the first chapter, precision and recall are metrics that explain relations between some selection of items and total items. This is the parameter that defines the size of said selection. This is how many items we will mark as “relevant” for each user and how many items we will request from our recommender
- `relevanceThreshold` – to distinguish between relevant and irrelevant items, we need some way of identifying said relevant items. This parameter specifies a minimum rating an item must receive to be considered relevant. In addition, it is possible to pass a special parameter `CHOOSE_THRESHOLD`. This parameter tells the evaluator that we want it to decide on a threshold on its own based on user’s ratings.

When the `evaluate` method is invoked, much like in the previous subsection’s evaluator, it

iterates through the users present in the data model and chooses users to use for evaluation at random, using a random number generator and comparing its output with `evaluationPercentage`.

Then, if we have not provided a specific threshold, one gets calculated for the current user. To do this, Mahout calculates average value of user's ratings and adds one standard deviation. This might be useful if users have many different rating patterns. For example, some user might use the whole scale, while another only using the extremes. Calculating individual threshold for each user can help mitigate these differences in rating patterns.

Next step is splitting ratings between training and test preference collections. In order to do that, for every user picked for testing, we go through his or her ratings until we encounter `at` items that the user rated higher then the threshold we use. These items are then saved in test preference collection with all the other items being put in training preferences.

After splitting is finished, training preference collection is used to create a data model for the recommender being evaluated. The recommender is then given testing user IDs one by one and ask to recommend `at` items. We then compare the selection provided by the recommender and the selection we predetermined while splitting ratings using a threshold and calculate precision and recall.

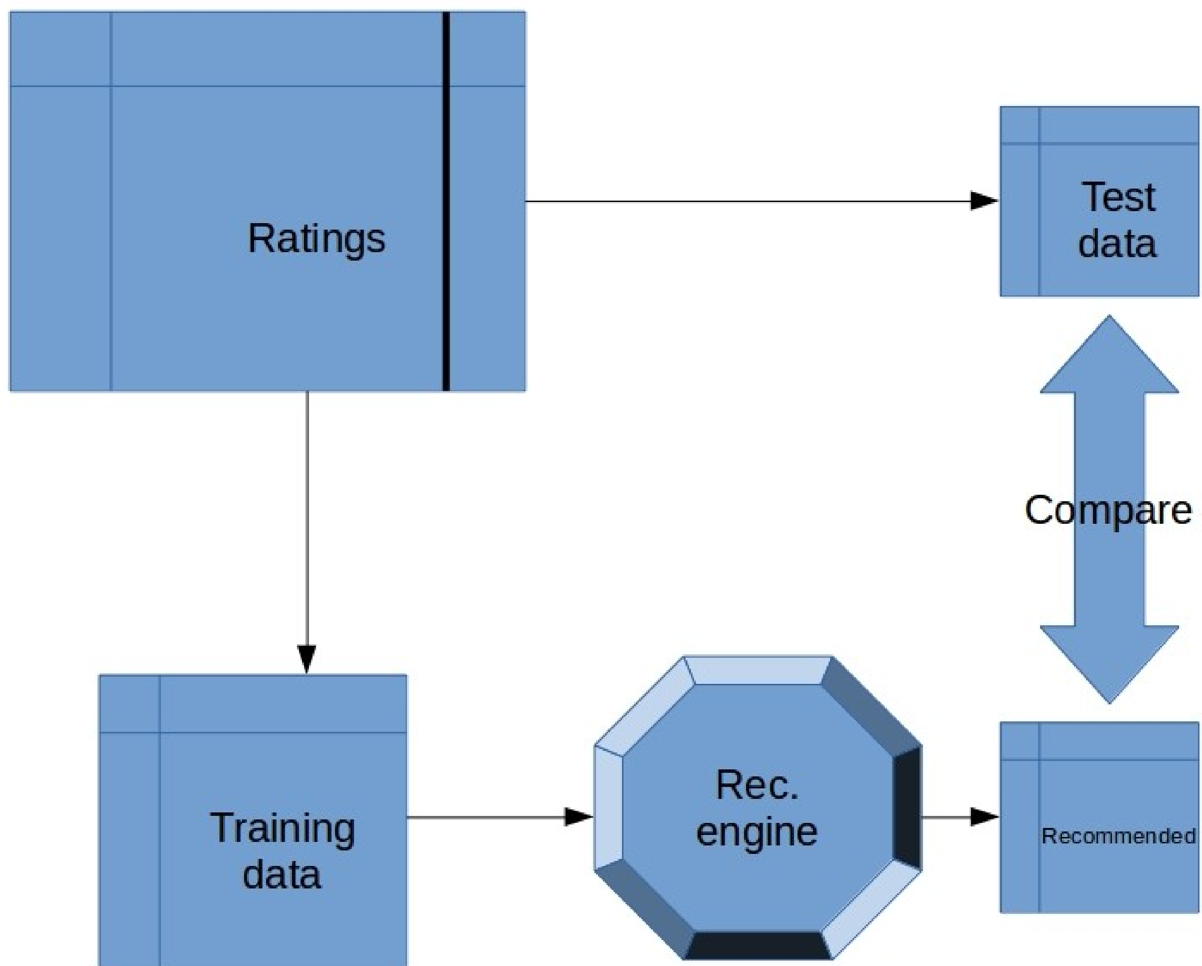



*Figure 8: Basic flow of the evaluation process*

After recommender has attempted to offer recommendations to every test user, the average values of resulting precision and recall values are presented as a final result of the evaluation process. These values can then be used on their own, or used to manually calculate F-score.

With Mahout-specific background out of the way, I will now explain the group-centered recommendation specific changes that I have made to the Mahout's process.

## 3.2 Experiment design

### 3.2.1 Evaluating predicted rating accuracy

First difference between Mahout's single-user recommender evaluator and my multiple-user recommender evaluator begins at the `evaluate` method signature:

```
evaluate(RecommenderBuilder recommenderBuilder,
         DataModelBuilder dataModelBuilder,
         DataModel dataModel,
         double trainingPercentage,
         double evaluationPercentage,
         int maxGroupSize,
         GroupDataProvider groupDataProvider)
```

If we compare this to Mahout's `evaluate` method we will notice that parameters named `maxGroupSize` and `groupDataProvider` has been added. `MaxGroupSize` is the parameter that tells us, how big we want the testing (and training) groups to be. `GroupDataProvider` will be discussed at length a little below.

It should also be noted that the `evaluate` method accepts a `RecommenderBuilder` and not a `GroupRecommender`. It is therefore theoretically possible to provide a builder that is made to build on of Mahout's stock single-user recommenders. This will however lead to a `ClassCastException` being thrown as TestSuit expects the builder to return a `GroupRecommender`.

When the `evaluate` method is invoked, much like with Mahout, we start by walking through the user IDs and picking them out at random. Next step, however, is different. We then attempt to find a group for the user.

This is achieved using an implementation of the aforementioned `GroupDataProvider` interface. This interface defines a single method `getGroupForUser(int userId, int howBig)` where

- `userId` – ID of the user, for whom we are creating a group
- `howBig` – the desired size of a group

This has been created as a separate interface because in the current version of the TestSuit searching for a group is highly tied to the system being tested. My implementation of this interface makes the following call to the database using jOOQ:

```
create.select(RATING.USER_ID, count)
      .from(RATING)
      .where(RATING.GAME_ID.in(
        create.select(RATING.GAME_ID)
```

```
        .from(RATING)
        .join(USER_ACCOUNT)
        .on(USER_ACCOUNT.ID.equal(RATING.USER_ID))
        .where(USER_ACCOUNT.ID.equal(userId)))
    ).groupBy(RATING.USER_ID)
     .orderBy(count.desc())
     .limit(howBig)
     .fetch(0, Integer.class);
```

Thanks to jOOQs syntax anyone that can read SQL should be able to easily understand what transpires here. I select user IDs of `howBig` users that have rated the most similar collection of games as the user specified by `userId`.

This particular method of group selection was chosen to emulate more or less stable groups that frequently try new games together as this is the target audience of the Sibyl application. I have also considered taking friendship status information into account but have ultimately decided to leave that, as well as trying out other methods of selecting groups that might come up, to future work.

As is evident from the code listing, my particular implementation of `GroupDataProvider` is only really able to get groups from the database that is exactly like the one Sibyl uses. This, of course, is not very useful to anyone but me. This is the reason why I have ultimately decided to create the `GroupDataProvider` interface and leave its implementation to the end user of TestSuit.

This does mean that some amount of work must be done by the person willing to use the TestSuit. However, it frees the user to implement the group selection method that he or she might be most interested in. For instance a family-targeted recommendation site might be more concerned with recommending things to groups consisting of people that are connected by family ties as opposed to groups that have tried many different items together. Their desired group choosing method will then consist of checking their database for all relatives of the user provided and forming a group out of them.

When a group is generated for testing, it then has to have their ratings separated between actual and test datasets. To do that, we first make a map object, that maps user IDs of the group members to lists of their preferences. We then make an extra copy of the list of preferences of the user whose ID we have initially used to generate a group as “seeding” preferences.

We then go through the seeding preference array one by one and, in a way similar to Mahout's own evaluators, start generating random Double values to correspond with each of the user’s preferences. If a random value is smaller then the `trainingPercentage` we put the preference that it was generated for into the training preference collection for the initial user.

If the generated random Double value is larger then `trainingPercentage`, the preference the value was generated for is put into test data preference array for the initial user. Next, all the other group member preference arrays are scanned for a rating on the same item and, if found, it is put into the respective users’ test preference arrays, forming a collection of test preference arrays for the group. This is done to emulate the situation when the whole group have not yet tried and rated an item as in an opposing event the group is more likely to just discuss the item between themselves and are less in need of a recommendation.

If a rating for an item is not found, it is replaced with an `EmptyPreference` placeholder object that defines a preference as NaN. This is needed to mitigate some of Mahout's particularities when dealing with users that don’t have any preferences.

Lastly, we scan all the non-initial group members' preference arrays and copy ratings on items that are not present in test preference arrays of the group members into each of the respective users' training data.

After splitting is done, we add all the user IDs of the group to the set of used IDs. This set is consulted every time a new user is picked out for testing and if he is already present in the set we ignore him. This is done so that test data would not be plagued by duplicate groups thus making results wildly inaccurate.

After the splitting process is finished, we form a new data model using the training preferences collected. This training model is then used to make a new group recommender and the evaluation process begins.

To evaluate the recommender we take our testing groups one by one and try to estimate a score for every game that is present in the testing data. After we get an estimation for a game, we apply the RMSE formula presented in section 1.2.5. The resulting value is then stored until all the tool goes through all the test data and attempts to estimate preference for every game in test data for every test group. An average of all the RMSE results is calculated and presented as the ultimate evaluation result.

### 3.2.2 Evaluating precision and recall

As with the evaluator discussed in previous subsection, the `evaluate` method for the evaluator doing precision and recall calculation in TestSuit is similar-yet-different to that of Mahout's `RecommenderIRStatsEvaluator`:

```
evaluate(RecommenderBuilder recommenderBuilder,
         DataModelBuilder dataModelBuilder,
         DataModel dataModel,
         IDRescorer rescorer,
         int at,
         double relevanceThreshold,
         double evaluationPercentage,
         int maxGroupSize,
         GroupDataProvider groupDataProvider)
```

Differences are similar in nature to what was described in previous subsection so I will not repeat myself.

The evaluator begins, again, with picking out random users by comparing Double values produced by the Random Number Generator with `evaluationPercentage`. We then consult the `GroupDataProvider` to determine a group for the user.

What we do next is attempt to generate a group of `at` relevant items for the whole group. To do that we first calculate a separate relevance threshold for every user if one was not provided via arguments. Calculation is the same as in Mahout's evaluator.

We then go over the "seeding" user's ratings, much like in Mahout to find an item that has a rating above our threshold. After we find the item, we check what ratings have other users given to the item. If every other user in the group has also rated the item above their respectable thresholds (or the one provided) the item is then added to the relevant item list for the group.

One might already see some potential problems that might arise with this design. With every increase in group size the likelihood of meaningful amount of relevant items drops dramatically. Same goes for the increasing `at` value. Unfortunately, given the nature of the problem at hand,

these issues do seem almost impossible to mitigate fully and the only advice I can give on the matter is to choose groups that have rated a large amount of similar items and keep the `at` value reasonably low.

After we have splitted away the relevant items, we form a training data model for our recommender using all the rest of the data we have. Then we attempt to recommend `at` number of items to every group and see how successful our recommender was at recommending relevant items. The precision and recall values resulting from the evaluation of every group are stored and the average values are then presented as a result of the evaluation.

### 3.2.3 The experiment setup

With the design of the evaluators out of the way we may now move on to the description of the experiment itself.

My testing database is a snapshot of BoardGameGeeks database and contains information on 76475 games and 141050 users that have generated 7625412 ratings. As much as I would love to test the system using the whole dataset, evaluation is quite a lengthy process that, if using `evaluationPercentage = 1` might take days to complete. That is why I had to choose a much smaller value.

I have decided to use `evaluationPercentage = 0.2` and `trainingPercentage = 0.8` for all evaluator runs. This means that every time we have taken about

$$141050*0.2=28210$$

total users and divided them into groups of varying size. This, in my opinion, provides a reasonable trade-off between having a large enough testing user base for the results to remain meaningful and not waiting several weeks for all the tests to complete.

Predicting the number of ratings is a bit trickier, because we do not know, how many ratings our chosen users have. However, if we take the unluckiest selection possible (all 28210 people are users with the least ratings), we can get a lower bound. My database contains 17624 people with only 1 rating, 7092 with 2 and 5158 with 3. This would mean that in the unluckiest choice possible, we would get all people with 1 and 2 ratings and 3494 with three. This lives us with

$$17624+7092*2+3494*3=42290$$

ratings to work with, out of which 8458 we will be trying to guess and the rest will be used as training.

Calculating the upper bound (the most lucky pick in terms of rating quantity) seems quite impractical, as there are 225 people in the database with more then a thousand ratings each. Out of them, 19 have several thousand each. Adding all of their rating quantity one by one (as they all have a unique number of ratings) would take too much space.

We could produce a rough estimate of the average expected number by assuming that 20% of users making it into evaluation bring with themselves 20% of ratings. That would mean, that our whole evaluation data contains

$$7625412*0.2\approx 1525082$$

ratings, of which 305016 (both numbers rounded down) we will be trying to guess.

So far, I had only been able to perform experiments with the predicted rating accuracy evaluator. So, for every experiment we

1. take the whole user base and choose 20% of users at random
2. find a user group for each user with group size provided at the start of the run
3. split away 20% of test users' ratings
4. attempt to predict the ratings we split away using a recommender provided at the start of the run
5. calculate the RMSE for every prediction comparing it to actual values saved in the test data
6. store all the calculated RMSE values
7. calculate the average of all RMSE values as a final evaluation

#### 3.2.4 Configurations being tested

With our recommender and evaluator we can easily test different combinations of differently configured recommenders and aggregators. The only constraint is the time spent evaluating as going through all the data does tend to take quite long.

I will now define different recommender-aggregator combinations that I have chosen.

##### 3.2.4.1 The random recommender

First one is `RandomRecommender` as a base single-user recommender and `AverageAggregator` as an aggregator.

`RandomRecommender` is one of the standard Mahout recommenders and operates in exactly the way that one would expect from a recommender with such a name. It produces random values and returns them as estimates.

I have decided to start with `RandomRecommender` to produce a baseline that other recommenders should surpass to be considered useful. In addition to that, it would be a good test for the TestSuit. It would seem obvious that any kind of thought-out algorithm must produce a better result than pure random. Especially since we are using recommenders similar to those that were tried out and were demonstrated to produce satisfactory results in previous studies on the topic of group-centered recommendation systems.

This means that if the TestSuit will be able to tell that the other recommenders produce more accurate results then a group recommender that uses `RandomRecommender` as a base recommender, we can trust that the evaluations it produces do have value in determining the best recommender configuration.

It does not really matter which aggregator to use with this particular recommender as results are random regardless. I have taken the `AverageAggregator` simply because this was the first aggregator that I had implemented.

##### 3.2.4.2 User neighborhood based with average aggregator

Second configuration being tested is the one that is expected to produce actual results and achieve estimation accuracy that is above the baseline. The configuration is using Mahout's `GenericUserBasedRecommender` as base. The base recommender is configured to use a similarity metric that is represented by Mahout's `CityBlockSimilarity` class.

`CityBlockSimilarity`, as is evident from the name, implements similarity metric called City Block Distance (also known as Manhattan distance). Similarity between two users or items with this similarity metric is calculated using the formula

$$d_{ij}=\sum_{k=1}^{n}|x_{ik}-x_{jk}|$$

where $d_{ij}$ is the City Block Distance itself. The smaller this number is the more similar the users or items being evaluated (denoted by $i$ and $j$ ) are to each other. $n$ refers to the number of ratings the items or users being compared have in common. $x_{ik}$ and $x_{jk}$ are the ratings users $i$ and $j$ have given to the k-th item in user-based neighborhood model or the ratings items $i$ and $j$ have received from the k-th user in item-based neighborhood model.

As a neighborhood model, the base recommender uses the model implemented in Mahout's `NearestNUserNeighborhood` class. As is evident from the name, this class instructs the recommender to use ratings of N users who are "nearest" to the user receiving recommendations according to the similarity metric the recommender uses.

I have set N to be 100. This is somewhat arbitrary and rerunning the evaluator with N set to other values to find an optimal one is certainly a possible direction for future work.

The aggregator being used in this configuration is the same as in the previous configuration: the `AverageAggregator`. It is the simplest and most basic aggregator imaginable. It takes in a list of values and calculates their average. This average is then returned as the aggregation result.

The main thought behind this configuration is twofold. On one hand, it is designed to be a trial for the TestSuit to see if it can recognize the difference between a random recommender and the recommender that actually has a method behind its estimations. For this task even the most simple and basic recommender would suffice. The second thing I wanted to do is to try and compare different aggregators. So I assumed that keeping the base recommender simple and basic would make the differences in aggregator choice more prominent.

#### 3.2.4.3 User neighborhood based with least misery aggregator

The configuration of this recommender's base recommender is identical to the previous one. Again, it uses `GenericUserBasedRecommender` as a recommender class, `CityBlockSimilarity` as a similarity metric and `NearestNUserNeighborhood` with N set to 100 as a neighborhood model.

The difference comes from the aggregator used. For this configuration I have introduced a new aggregator class called `LeastMiseryAggregator`.

The `LeastMiseryAggregator` uses an aggregation strategy similar to that of the group-centered recommendation system PolyLens[3]. This aggregation strategy, called Least Misery aggregation strategy, operates under the assumption that a group enjoys an item as much as the least satisfied group member.

In accordance to this assumption, when presented with a collection of ratings, the aggregator returns the lowest rating as an estimate for the whole group. That way, as the reasoning behind this aggregation strategy goes, all the users in a group can expect a minimum degree of satisfaction that is equal to the estimated preference.

#### 3.2.4.4 Group sizes

The last bit of experiment design that I would like to explore is the size of the group that we use to perform our experiments.

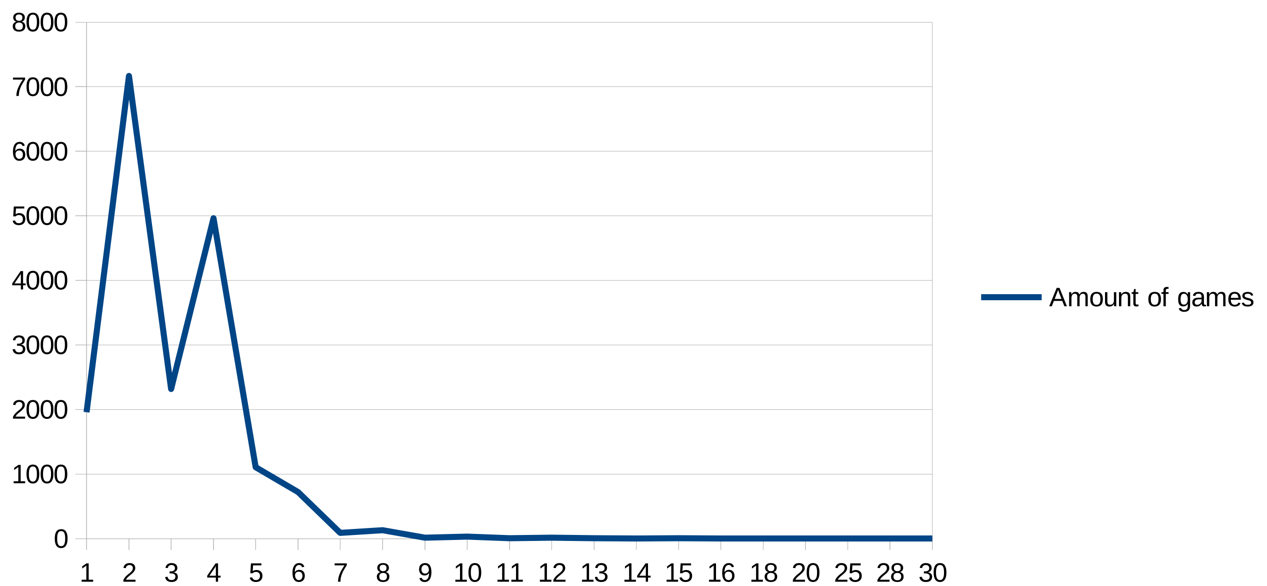
From my personal experiences in the board gaming community, the most common stable group size tends to be five people or less. We can also check the BoardGameGeek database to see, how many

games are best geared to different group sizes. To do this, we select all the elements in the `game` table and group them by the `bestplayers` column values, which represent the BoardGameGeek community's opinion on how large should be a player group to get the best gaming experience out of the given game.

The results are as follows:

| `Bestplayers` value | Games amount |
|---|---|
| 2 | 7168 |
| 4 | 4964 |
| 3 | 2318 |
| 1 | 1956 |
| 5 | 1106 |
| 6 | 722 |
| 8 | 128 |
| 7 | 90 |
| 10 | 29 |
| 9 | 13 |
| 12 | 13 |
| 11 | 8 |
| 13 | 5 |
| 15 | 4 |
| 30 | 3 |
| 16 | 2 |
| 18 | 1 |
| 14 | 1 |
| 28 | 1 |
| 20 | 1 |
| 25 | 1 |

The same table can be presented in a form of a graph:




(Note that not all of the games in the database have a `bestplayers` value, so if you will add up all the numbers in the games amount column, the result will be smaller then the amount of games declared earlier)

As we can see from the provided table and graph, board game designers also tend to operate under the assumption that most of the groups playing their game will have five or less players, so, consequentially, these are the groups that will interest us the most when providing recommendations.

This is why I have chosen to run an evaluation of every presented configuration four times, using group size of five, four, three and two.

With all the particularities of the experiments outlined, we may now move on to the experiments themselves.

## 3.3 Experiment 1: the random recommender

To demonstrate the experiment, I will now present and discuss some snippets of the output log of the TestSuit running the evaluation of the `RecommendationAggregatingRecommender` that has been constructed using `RandomRecommender` as a base recommender and `AverageAggregator` as an aggregator. More complete log example can be found in the appendix (page 49). All presented logs have been generated as output while running the actual experiments.

First, we shall look at the start and the end of the evaluation process

```
16/05/01 06:33:14 INFO
eval.AbstractDifferenceRecommenderEvaluator: Starting timing of
21893 tasks in 8 threads

...

16/05/01 06:33:16 INFO evaluators.AggregateEvaluator: Evaluation
```

```
result: 2.4924774956785107
```

As is evident from the time stamps, the evaluation itself has been quite quick, taking merely seconds.

```
16/05/01 06:33:15 INFO eval.StatsCallable: Average time per
recommendation: 13ms

...

16/05/01 06:33:15 INFO eval.StatsCallable: Average time per
recommendation: 0ms
```

Here, the evaluator reports average times per recommendation as almost instantaneous (13 to 0 milliseconds). This is, of course, quite predictable, as the recommender here does not do any actual work and just fires away random numbers.

Interestingly, the aggregator also does not seem to impact the system so much, which could also be taken as good news for the future of Sibyl performance-wise.

What does take quite some time, however, is the formation of groups and splitting of ratings between the training preference and test preference collections. On average, the splitting for the random recommender took between two and three hours:

```
====Beginning evaluation of random recommender with
AverageAggregator====

16/05/01 03:52:40 INFO evaluators.AggregateEvaluator:1001 users
processed in splitting

...

16/05/01 06:33:14 INFO evaluators.AggregateEvaluator: Beginning
evaluation of 21893 users

16/05/01 06:33:14 INFO
eval.AbstractDifferenceRecommenderEvaluator: Starting timing of
21893 tasks in 8 threads
```

Optimizing this part of the process seems to be a good candidate for future work. Candidate solutions that I have in mind might include, for instance, removing the need of database calls to calculate the group IDs and making my own classes instead of using Mahout's ones that would make manipulating group data more streamlined and efficient.

The results of test runs are as follows:

| Group size | Evaluation result |
|---|---|
| 5 | 2.4925 |
| 4 | 2.5564 |
| 3 | 2.6212 |
| 2 | 2.7540 |

I would like to leave the detailed discussion of results of the evaluation to the Discussion subsection (page 44). One thing that is immediately noticeable, however, is that the scores of the random

recommender become noticeably smaller with group size increasing.

This trend is, of course, not entirely surprising. With the growth of the group size, the likelihood of a random prediction being close enough to an actual rating by any of the users also grows.

While not being surprising, this trend does offer some interesting implications. First one being that we should focus on precision of rating predictions for smaller group sizes as they seem to be harder to get right “on accident”. The second one is that as group size grows, so do the expectations we have for our recommender.

It is also entirely possible that if we continue to expand the group size, the trend will at some point reverse. I, however, believe that this is quite irrelevant to the problem at hand since, as stated above, board gaming groups that are larger then five people are not very common in real world. And even if we ignore the fact that the focus of this work is recommending board games and consider a general purpose group-centered recommendation system, stable groups of people that tend to do things together are rarely much larger then the ones we already have covered.

## 3.4 Experiment 2: user-based recommender with average aggregator

As with the previous experiment, I shall now demonstrate the TestSuit log snippets for running an `RecommendationAggregatingRecommender` that has been constructed using `GenericUserBasedRecommender` as a base recommender. The recommender uses `CityBlockSimilarity` as a similarity metric and `NearestNUserNeighborhood` with N set to 100 for determining the neighborhood. `AverageAggregator` is used as an aggregator.

Splitting has, again, taken quite some time. In this particular case creation of testing groups and splitting their preferences has taken a little less then two hours:

```
16/05/01 23:00:47 INFO evaluators.AggregateEvaluator: 1000 users processed in splitting

...

16/05/02 01:46:23 INFO evaluators.AggregateEvaluator: Beginning evaluation of 21797 users
```

The most striking difference, however, is between the time the random recommender needed to go through the evaluation process and the time our current recommender took.

```
16/05/02 01:46:23 INFO eval.AbstractDifferenceRecommenderEvaluator: Starting timing of 21797 tasks in 8 threads

...

16/05/02 04:56:08 INFO evaluators.AggregateEvaluator: Evaluation result: 1.3507994504960197
```

As timestamps clearly show, it took our recommender slightly under three hours to go through all the test data and perform the evaluation.

It should now be clear why I have chosen to use only 20% of users as test data as opposed to using the whole database or, at least, a bigger percentage. With user base growing, the time it would take for both splitting and evaluating would grow even larger. As one of the tests I have tried running an

evaluation on full user base. It was left running for a whole day and was not anywhere near finishing when I came back to check on it.

While fixing the splitting performance would certainly help with the problem, I have my doubts about anything being doable for recommendation step in this regard. I have tried running Mahout's single-user recommender on the whole database too and it took several days for it to finish.

Mahout is a mature and actively developed product so I sincerely doubt that this is a case of poor optimization. More likely, the nature of the task, the sheer amount of data points that need to be processed, even if calculation needed for one data point on it's own is fairly small, amount to a lot of work. So much work, in fact, that my Intel(R) Core(TM) i7-4770K CPU @ 3.50GHz that I have used as the machine to run the evaluators on, was not enough to complete the task in a reasonable amount of time.

I am afraid the only solution I can offer to speed up the evaluation process is "throw more cores at it", as in, create or gain access to a, preferably, industry-grade cluster. With this kind of resources, I would imagine that evaluating even the full database would be a matter of minutes.

Obviously, I do not have these kind of resources at my disposal, so I had to make do with what is available to me.

The results of the test runs are presented below:

| Group size | Evaluation result |
|---|---|
| 5 | 1.3508 |
| 4 | 1.2682 |
| 3 | 1.2235 |
| 2 | 1.1748 |

First thing to notice here is that TestSuit has been successful in identifying this recommender as more precise. The values presented are almost two times smaller then the ones the random recommender has demonstrated. This is good news, as it tells us that TestSuit's evaluation does indeed reflect reality and may offer us some value when further evaluating recommenders.

Another thing to note is that the trend we have noticed in random recommender evaluations is reversed here, although not so dramatically: the accuracy of predicted ratings grows a little with the group size shrinking.

This effect is also not entirely unpredictable. With group size growing producing a prediction that is indicative of every group member's preference becomes increasingly hard, as anyone who tried to pick a restaurant for a group of picky eaters could confirm.

Despite that, I believe that the results above indicate this particular configuration a good starting point for a group-centered recommendation system and, while not perfect, it could definitely provide it's users some good value.

It does however indicate that rating precision is an issue with larger groups and attempts at combating that could prove to be a promising direction for future work.

## 3.5 Experiment 3: user-based recommender with least misery aggregator

Similarly to previous two experiments I shall now present snippets of TestSuit's output log for running an `RecommendationAggregatingRecommender` that has been constructed using `GenericUserBasedRecommender` as a base recommender. The recommender uses `CityBlockSimilarity` as a similarity metric and `NearestNUserNeighborhood` with N set to 100 for determining the neighborhood. `LeastMiseryAggregator` is used as an aggregator.

```
====Beginning evaluation of User based recommender with
LeastMiseryAggregator====

16/05/02 22:12:59 INFO evaluators.AggregateEvaluator: Beginning
evaluation using 0.8 of
org.apache.mahout.cf.taste.impl.model.jdbc.ReloadFromJDBCDataModel
@e720b71

...

16/05/03 01:20:48 INFO evaluators.AggregateEvaluator: 31000 users
processed in splitting

16/05/03 01:20:48 INFO evaluators.AggregateEvaluator: Beginning
evaluation of 21892 users

...

16/05/03 04:51:39 INFO evaluators.AggregateEvaluator: Evaluation
result: 1.358727842549387
```

Similarly to the previous configuration, both the splitting and the evaluation took quite some time. I have already discussed this issue when talking about the previous experiment configuration, so I am not going to repeat myself right now.

The results of the experiment are presented below:

| Group size | Evaluation result |
|---|---|
| 5 | 1.3587 |
| 4 | 1.2836 |
| 3 | 1.2134 |
| 2 | 1.1743 |

As we can see, least misery aggregator have also managed to beat the random by a noticable degree. We may also notice that the trend of recommendation precision growing for smaller user groups that was also present in the average aggregator experiment.

What is different, however, is that this precision growth is noticeably slower then in case of the average aggregator. This seems quite predictable as this particular aggregator does effectivly return one's user preference, while disregarding all the others as long as they are higher.

I would not, however, oughtright dismiss the least misery aggregator as it still may have its benefits

if taking precision and recall into consideration. More on that in the discussion subsection.

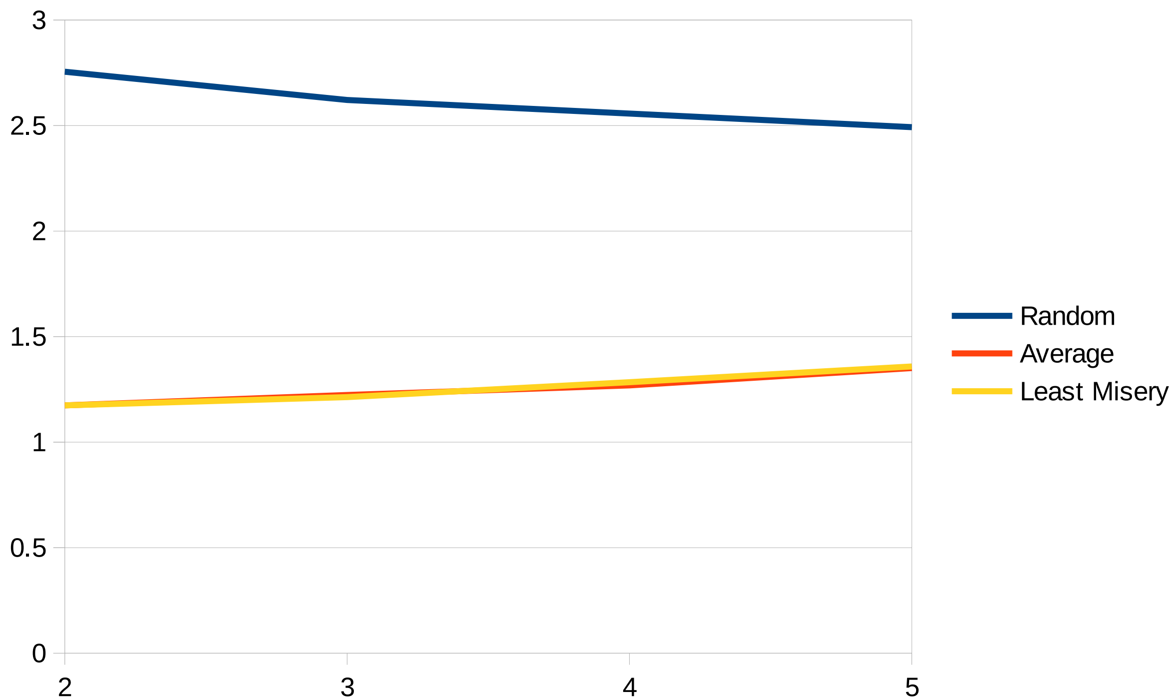
## 3.6 Discussion

With all the experiments completed and all the data collected, let us now look at it once more and discuss, what can we conclude from the data gathered.

For the sake of convenience I shall combine all the experiment results in one single table presented below:

| Group size | Random recommender | User-based recommender with average aggregator | User-based recommender with least misery aggregator |
|---|---|---|---|
| 5 | 2.4925 | 1.3508 | 1.3587 |
| 4 | 2.5564 | 1.2682 | 1.2836 |
| 3 | 2.6212 | 1.2235 | 1.2134 |
| 2 | 2.7540 | 1.1748 | 1.1743 |

Let us also look at the same data in a form of a graph to visualize the trends more clearly




First thing we can clearly notice is that both of our recommenders have successfully beaten the random by a significant margin. This would indicate, that the recommenders we constructed are indeed capable of providing users of the system valuable recommendations that they are likely to enjoy.

The presented error is still quite high compared to errors usually presented in single-user recommender evaluations. When I was playing around with single-user recommenders and the database, numbers that I got were almost always smaller then one. This is an unfortunate consequence of the lack of guarantee that all the users enjoy an item equally.

If a group of five people has rated an item as [10, 9, 10, 8, 2] it is virtually impossible to produce a suggested rating that would not be inaccurate for at least one user of the group. Striving to produce such a rating, however, should not be considered a waste of time, because users consulting such ratings can still get enough information for an informed decision.

For example, if we take the average aggregator, the aforementioned rating set would produce

$$\frac{10+9+10+8+2}{5}=7.8$$

However an item that would receive ratings [8, 7, 9, 8, 10] would receive the predicted rating of

$$\frac{8+7+9+8+10}{5}=8.4$$

As you can see, while the first item would be much more enjoyable for all but one user of the group, the item where users' satisfaction can be considered unanimous, and therefore, is closer to the goal of pleasing every member of the group without anyone feeling left out, has higher predicted rating. And as such, the users consulting the system are more likely to pick that item and be satisfied with it on a more-or-less same level.

Second thing of interest is the relationship between group size and precision. Here we can note that the random recommender is almost magically becoming more precise as the group size grows, while the proper recommenders are losing accuracy. This is, of course, due to the fact that the more people are in the group, the more likely we are to produce a prediction that some of them will find reasonable (an effect that people pretending to be psychics love to abuse when working with crowds). However, the more people there are in a group, the less likely we are to produce a prediction that most of them will find sensible.

If I were to extrapolate, I would predict that proper recommenders' and random recommender's accuracy would approach each other asymptotically with growing group sizes. Thankfully, the size of a group, where the difference between a proper recommender and a random one becomes irrelevant, is too large to be of interest in any realistic setting. Then again, sometimes it is just impossible to please everyone.

This allows us to make a conclusion that our system is indeed capable of producing relevant and valuable recommendations.

As for the comparison of our aggregators' performance, they have shown quite comparable results with average aggregator being slightly more precise with larger group sizes but losing to the least misery aggregator on the smaller ones. This would suggest, that it is more beneficial to use the average aggregator for groups that have four or more people and the least misery for the smaller ones when calculating predicted ratings to display.

I would, however, advise against dismissing least misery aggregator altogether when dealing with large groups. As we have not actually measured precision and recall (discussed on page 15) it is not out of the question that by those metrics least misery aggregator will be able to outperform average aggregator on every group size.

If we define relevant item as "item that has above-threshold ratings for all users" it is clear that least

misery will be able to filter out such items more effectively as average aggregator can presume that an item has high predicted score if group has only one user that is unlikely to like the item with all others being likely to love it.

If we were to run such an experiment and my hypothesis is proven correct, I could imagine a “double filtering” system, similar to “average without misery” strategy proposed in [12]. In such a system, relevant items would first be chosen by least misery aggregator and then predicted rating for displaying to user is calculated using whichever aggregator is more precise for the group size at hand.

One might wonder, however, about the random aspect of our testing. How trusting can we be of our results? Could an aggregator simply get lucky and have gotten a better score by chance? One method to test for this and, indeed, the one Mahout suggests for single-user recommenders, is to run the test multiple times and see which recommender shows better results on average.

This method, however, did not suit me because of just how long the whole evaluation process takes. So I have chosen a different approach: I have reinitialized the random number generator the evaluator uses before each evaluation. While doing that, I have also provided the same seed to the generator.

What this means is that for this next test run, both aggregators have received exactly the same user set with exactly the same ratings splitted away. What’s more, because only the “seeding” user and item are chosen for group creation and splitting, groups and ratings were similar between two runs of the same aggregator too.

Here are the result of this randomless run:

| Group size | User-based recommender with average aggregator | User-based recommender with least misery aggregator |
|---|---|---|
| 5 | 1.3162 | 1.3250 |
| 4 | 1.2790 | 1.2871 |
| 3 | 1.2249 | 1.2284 |
| 2 | 1.1904 | 1.1903 |

As we can see, both tendencies that we have noticed in the previous run are still present, if a little less prominent. Prediction accuracy is still dropping with the group size and least misery is still ever-so-slightly better on groups with smaller sizes.

# 4. Conclusions and future work

In this work I have created three tools: BGGParser, Sibyl and TestSuit.

Of these three, Sibyl and TestSuit have been the main focus of the work. First one – as a prototype of a potentially first ever group-centered collaborative filtering tool for recommending board games and, more importantly, first such tool that is aimed to be deployed and used in a production environment.

TestSuit has been demonstrated to be a useful tool for evaluating group-centered recommendation engines in terms of ratings prediction accuracy.

We have also run tests that demonstrate that

- Sibyl’s current recommender is absolutely capable of beating random recommendations in terms of prediction accuracy and is therefore capable of providing valuable and relevant recommendations to Sibyl’s users
- Average aggregator is generally slightly better for providing estimated ratings for groups then the least misery aggregator in terms of accuracy on groups with 4 and more members, so if we are interested in presenting accurate ratings that the user group will find believable, average aggregator should be preferred over the least misery aggregator on large groups and vice versa – on small ones

## 4.1 Future work

As both Sybil and TestSuit are pretty much in the infancy stage, there is a lot to be done as possible future work. I have noted all the possible directions for moving forward when they were relevant, but I will collect them all and reiterate here for reader’s convenience.

Here are all the directions for future work that I have managed to identify:

- In this work we have only explored the recommendation model that aggregates the predictions given for individual users. Exploring the virtual user alternative, running evaluations on it and comparing its performance to the model we employed is most certainly an interesting topic for future work.
- Comparing different solutions already present. It would most certainly be interesting to pit the solutions that already have been suggested by different researchers against one another. Unfortunately, these systems are not usually openly available. This means, that individuals that would wish to take upon themselves this particular direction should be ready to contact said researchers directly with requests for collaboration.
- Using metadata to fine-tune recommendations. In single-user collaborative filtering field this particular direction of work has been widely applied and group-centered collaborative filtering should not be left behind. Board games in particular offer a very promising natural way of filtering out results via minimum and maximum numbers of players as well as a BoardGameGeek provided “recommended” number. Incorporating use of this data in filtering is a good way to boost recommendation relevance, which is why this particular item is quite high to the top in Sibyl’s feature backlog.
- Producing groups for testing. This particular part of the evaluation process certainly needs some work to be able to produce more realistic and relevant user groups. Starting from performance improvements, that could consist of moving the whole group selection process in memory, to more advanced selection strategies (eg are users friends? Do they live in the same city?). This work could not only speed the evaluator up considerably, but also would help produce more realistic and relevant testing data by making testing groups out of users who are more likely to belong to the same gaming group in reality.
- NaNs and aggregators. As you might remember, sometimes Mahout is unable to predict a rating for a particular item for a particular user. In such situations a NaN is returned as the estimated preference. This, of course, poses a question of dealing with them when aggregating estimated scores. There could be different ways to deal with NaNs. Treat them as the “average” score (3 in 5 point system, 5 in 10 point system etc), ignore them

altogether, treat them as minimum score, treat them as maximum score etc. I have chosen to go with the "ignore them" approach. It would certainly be interesting to try out other approaches and see what impact they would make on recommendation engine performance.

Same question could be raised for virtual user models of group-centered collaborative filtering. What do we do if out of three group members only two have rated an item? Different strategies for dealing with NaNs would also come into play in situations like this.

- Different base recommenders. Mahout provides a great variety of recommenders out of the box. These recommenders can then be further customized via different parameters, rescorers, neighborhood metrics, similarity metrics etc. This provides a wide variety of possible recommender configurations. This work has only scratched the surface of the rich selection of recommenders readily available by Mahout. It would most certainly be a great idea to try out more different recommenders and see if any of them perform better then the ones already presented in this work.
- It has become apparent in the experiments that the larger the group is, the less precise the estimations. Trying to enhance the precision of recommendations for bigger groups therefore would be an exciting problem to solve.
- Trying different aggregators. In this work I have only managed to try two different aggregation functions. There are many more others used and proposed in research papers. It would, naturally, be beneficial to try and compare them too.
- Running experiments to find precision and recall. While having some groundwork for these experiments in place, technical difficulties have not allowed me to finish the implementation and run the tests. This is quite unfortunate, as these tests can certainly provide a different angle on evaluating group recommenders. As I already mentioned when discussing least misery evaluator experiment, it is not out of the question that some recommendation engine configurations, while losing in terms of recommendation accuracy, might have quite an edge over competition in terms of precision and recall.
- Bringing Sibyl to production. Last but not least, as the whole point was to create a potentially production-ready solution, Sibyl needs further development to continue growing in this direction. Things like login system, ability to rate items in the application itself, so there won't be a need to scan BoardGameGeek database for profile updates, a less spartan user interface and much more. All these features are present in the feature backlog and will be implemented in the future.

# Bibliography

1: Goldberg David,Nichols David, Oki Brian M., Terry Douglas, Using collaborative filtering to weave an information tapestry, 1992
2: Bennett James, Lanning Stan, The Netflix Prize, 2007
3: O'Connor Mark, Cosley Dan, Konstan Joseph A., Riedl John, PolyLens: A Recommender System for Groups of Users, 2001
4: McCarthy Joseph F., Anagnost Theodore D., MusicFX: an arbiter of group preferences for computer supported collaborative workouts, 1998
5: Ardissono L., Goy A., Petrone G., Segnan M., Torasso P., Tailoring the Recommendation of Tourist Information to Heterogeneous User Groups, 2002
6: Paul Resnick, Neophytos Iacovou, Mitesh Suchak, Peter Bergstrom, John Riedl, GroupLens: an

open architecture for collaborative filtering of netnews, 1994
7: Gediminas Adomavicius, Alexander Tuzhilin, Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions, 2005
8: Sean Owen, Robin Anil, Ted Dunning, Ellen Friedman, Mahout in Action, 2011
9: Yehuda Koren, Robert Bell, Chris Volinsky, Matrix Factorization Techniques for Recommender Systems, 2009
10: McCarthy Kevin, Salamo Maria, Coyle Lorcan, McGinty Lorraine, Smyth Barry, Nixon Paddy, Cats: A synchronous approach to collaborative group recommendation, 2006
11: Zhiwen Yu, Xingshe Zhou, Yanbin Hao, Jianhua Gu, TV Program Recommendation for Multiple Viewers Based on user Profile Merging, 2006
12: Judith Masthoff, Group Recommender Systems: Combining Individual Models, 2010
13: Zhou Shandan, Research on recommendation for group users, 2013
14: Bernier Cédric, Brun Armelle, Aghasaryan Armen, Bouzid Makram, Picault Jérome, Senot Christophe, Topology of communities for the collaborative recommendations to groups, 2010
15: Schelter Sebastian, Owen Sean, Collaborative Filtering with Apache Mahout, 2012
16: Aranda Jorge, Givoni Inmar, Handcock Jeremy, Tarlow Danny, An Online Social Network-based Recommendation System,
17: https://trello.com/b/IOd88pXO/gameshelf, gameshelf.se trello board, 2016
18: https://boardgamegeek.com/, BoardGameGeek website, 2016
19: https://boardgamegeek.com/wiki/page/BGG_XML_API2, BoardGameGeek XML API documentation, 2016, https://boardgamegeek.com/wiki/page/BGG_XML_API2
20: https://flywaydb.org/, Flyway website, 2016

# Appendix: TestSuit log samples

**Random recommender**

```
====Beginning evaluation of random recommender with
AverageAggregator====

16/05/01 03:52:40 INFO evaluators.AggregateEvaluator:1001 users
processed in splitting

16/05/01 04:09:15 INFO evaluators.AggregateEvaluator: 2000 users
processed in splitting

...

16/05/01 06:22:40 INFO evaluators.AggregateEvaluator: 20002 users
processed in splitting

16/05/01 06:24:45 INFO evaluators.AggregateEvaluator: 21000 users
processed in splitting

...

16/05/01 06:33:14 INFO evaluators.AggregateEvaluator: Beginning
evaluation of 21893 users

16/05/01 06:33:14 INFO
eval.AbstractDifferenceRecommenderEvaluator: Starting timing of
21893 tasks in 8 threads

16/05/01 06:33:15 INFO eval.StatsCallable: Average time per
```

```
recommendation: 13ms

16/05/01 06:33:15 INFO eval.StatsCallable: Approximate memory
used: 312MB / 376MB

16/05/01 06:33:15 INFO eval.StatsCallable: Unable to recommend in
0 cases

16/05/01 06:33:15 INFO eval.StatsCallable: Average time per
recommendation: 1ms

16/05/01 06:33:15 INFO eval.StatsCallable: Approximate memory
used: 314MB / 384MB

16/05/01 06:33:15 INFO eval.StatsCallable: Unable to recommend in
0 cases

...

16/05/01 06:33:15 INFO eval.StatsCallable: Average time per
recommendation: 0ms

16/05/01 06:33:15 INFO eval.StatsCallable: Approximate memory
used: 343MB / 511MB

16/05/01 06:33:15 INFO eval.StatsCallable: Unable to recommend in
0 cases

...

16/05/01 06:33:16 INFO eval.StatsCallable: Average time per
recommendation: 0ms

16/05/01 06:33:16 INFO eval.StatsCallable: Approximate memory
used: 464MB / 565MB

16/05/01 06:33:16 INFO eval.StatsCallable: Unable to recommend in
0 cases

16/05/01 06:33:16 INFO evaluators.AggregateEvaluator: Evaluation
result: 2.4924774956785107
```

**Average aggregator**

```
6/05/01 22:36:28 INFO evaluators.AggregateEvaluator: Beginning
evaluation using 0.8 of
org.apache.mahout.cf.taste.impl.model.jdbc.ReloadFromJDBCDataModel
@e720b71

16/05/01 23:00:47 INFO evaluators.AggregateEvaluator: 1000 users
processed in splitting

16/05/01 23:17:40 INFO evaluators.AggregateEvaluator: 2000 users
processed in splitting

...

16/05/02 01:01:01 INFO evaluators.AggregateEvaluator: 12000 users
```

```
processed in splitting

16/05/02 01:08:30 INFO evaluators.AggregateEvaluator: 13000 users
processed in splitting

...

16/05/02 01:46:20 INFO evaluators.AggregateEvaluator: 30000 users
processed in splitting

16/05/02 01:46:23 INFO evaluators.AggregateEvaluator: 31000 users
processed in splitting

16/05/02 01:46:23 INFO evaluators.AggregateEvaluator: Beginning
evaluation of 21797 users

16/05/02 01:46:23 INFO
eval.AbstractDifferenceRecommenderEvaluator: Starting timing of
21797 tasks in 8 threads

16/05/02 01:46:27 INFO eval.StatsCallable: Average time per
recommendation: 1512ms

16/05/02 01:46:27 INFO eval.StatsCallable: Approximate memory
used: 372MB / 468MB

16/05/02 01:46:27 INFO eval.StatsCallable: Unable to recommend in
64 cases

16/05/02 01:53:59 INFO eval.StatsCallable: Average time per
recommendation: 3578ms

16/05/02 01:53:59 INFO eval.StatsCallable: Approximate memory
used: 676MB / 1051MB

16/05/02 01:53:59 INFO eval.StatsCallable: Unable to recommend in
10342 cases

...

16/05/02 03:56:09 INFO eval.StatsCallable: Average time per
recommendation: 4789ms

16/05/02 03:56:09 INFO eval.StatsCallable: Approximate memory
used: 588MB / 1051MB

16/05/02 03:56:09 INFO eval.StatsCallable: Unable to recommend in
169377 cases

16/05/02 04:03:40 INFO eval.StatsCallable: Average time per
recommendation: 4702ms

16/05/02 04:03:40 INFO eval.StatsCallable: Approximate memory
used: 493MB / 1051MB

16/05/02 04:03:40 INFO eval.StatsCallable: Unable to recommend in
179347 cases
```

```
...

16/05/02 04:45:37 INFO eval.StatsCallable: Average time per
recommendation: 4300ms

16/05/02 04:45:37 INFO eval.StatsCallable: Approximate memory
used: 739MB / 1059MB

16/05/02 04:45:37 INFO eval.StatsCallable: Unable to recommend in
234391 cases

16/05/02 04:51:29 INFO eval.StatsCallable: Average time per
recommendation: 4230ms

16/05/02 04:51:29 INFO eval.StatsCallable: Approximate memory
used: 627MB / 1059MB

16/05/02 04:51:29 INFO eval.StatsCallable: Unable to recommend in
242741 cases

16/05/02 04:56:08 INFO evaluators.AggregateEvaluator: Evaluation
result: 1.3507994504960197
```

**Least misery aggregator**

```
====Beginning evaluation of User based recommender with
LeastMiseryAggregator====

16/05/02 22:12:59 INFO evaluators.AggregateEvaluator: Beginning
evaluation using 0.8 of
org.apache.mahout.cf.taste.impl.model.jdbc.ReloadFromJDBCDataModel
@e720b71

16/05/02 22:37:56 INFO evaluators.AggregateEvaluator: 1000 users
processed in splitting

16/05/02 22:54:34 INFO evaluators.AggregateEvaluator: 2000 users
processed in splitting

...

16/05/03 01:00:15 INFO evaluators.AggregateEvaluator: 16000 users
processed in splitting

16/05/03 01:02:57 INFO evaluators.AggregateEvaluator: 17000 users
processed in splitting

...

16/05/03 01:20:45 INFO evaluators.AggregateEvaluator: 30000 users
processed in splitting

16/05/03 01:20:48 INFO evaluators.AggregateEvaluator: 31000 users
processed in splitting

16/05/03 01:20:48 INFO evaluators.AggregateEvaluator: Beginning
evaluation of 21892 users
```

```
16/05/03 01:20:48 INFO
eval.AbstractDifferenceRecommenderEvaluator: Starting timing of
21892 tasks in 8 threads

16/05/03 01:20:52 INFO eval.StatsCallable: Average time per
recommendation: 1486ms

16/05/03 01:20:52 INFO eval.StatsCallable: Approximate memory
used: 821MB / 1093MB

16/05/03 01:20:52 INFO eval.StatsCallable: Unable to recommend in
87 cases

16/05/03 01:28:32 INFO eval.StatsCallable: Average time per
recommendation: 3673ms

16/05/03 01:28:32 INFO eval.StatsCallable: Approximate memory
used: 896MB / 1152MB

16/05/03 01:28:32 INFO eval.StatsCallable: Unable to recommend in
10011 cases

...

16/05/03 02:59:50 INFO eval.StatsCallable: Average time per
recommendation: 5941ms

16/05/03 02:59:50 INFO eval.StatsCallable: Approximate memory
used: 541MB / 1161MB

16/05/03 02:59:50 INFO eval.StatsCallable: Unable to recommend in
121428 cases

16/05/03 03:11:30 INFO eval.StatsCallable: Average time per
recommendation: 5896ms

16/05/03 03:11:30 INFO eval.StatsCallable: Approximate memory
used: 346MB / 1161MB

16/05/03 03:11:30 INFO eval.StatsCallable: Unable to recommend in
133710 cases

...

16/05/03 04:39:22 INFO eval.StatsCallable: Average time per
recommendation: 4762ms

16/05/03 04:39:22 INFO eval.StatsCallable: Approximate memory
used: 956MB / 1161MB

16/05/03 04:39:22 INFO eval.StatsCallable: Unable to recommend in
241790 cases

16/05/03 04:46:06 INFO eval.StatsCallable: Average time per
recommendation: 4692ms

16/05/03 04:46:06 INFO eval.StatsCallable: Approximate memory
used: 812MB / 1161MB
```

```
16/05/03 04:46:06 INFO eval.StatsCallable: Unable to recommend in
250936 cases

16/05/03 04:51:39 INFO evaluators.AggregateEvaluator: Evaluation
result: 1.358727842549387
```