TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Computer Science

ITV40LT

Ivan Studenikin 120478IABB

ANALYSIS OF ALGORITHMS USED IN RECOMMENDER SYSTEMS AND USABILITY OF TOOLS IMPLEMENTING THEM

Bachelor's thesis

Supervisor: Ago Luberg
Master of Science
Assistant

Tallinn 2016

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Arvutiteaduse instituut

ITV40LT
Ivan Studenikin 120478IABB

SOOVITUSALGORITMIDE JA NEID RAKENDAVATE TÖÖRISTADE KASUTUSMUGAVUSE ANALÜÜS

Bakalaurusetöö

Juhendaja: Ago Luberg
Magister
Assistent

Tallinn 2016

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Ivan Studenikin

23.05.2016

Abstract

The main goals of this thesis are analysis of popular algorithms used in recommender systems, their presentation in an easy to digest manner and analysis of usability of software frameworks that implement these algorithms.

In the second chapter, we look into the most popular recommendation algorithms based on the work of researchers. They are analysed in terms of idea behind them, various features and types of the algorithms, performance, scalability, application and accuracy. The analysis is presented in a manner readable also for non-technical people.

In the third chapter the usability of three advanced frameworks implementing the algorithms, presented in the second chapter, is tested. It is done by building a simple recommender system, using the provided documentation for the frameworks.

Finally, in the summary conclusions are drawn as to which algorithms are suitable for different use cases and what framework is suitable for which application.

Analysis presented in the thesis should be sufficient to enable informed choice of algorithms and frameworks to be used in development of a domain specific recommender system.

This thesis is written in English and is 34 pages long, including 4 chapters, 11 figures and 1 table.

Annotatsioon
Soovitusalgoritmide ja neid rakendavate tööristade kasutusmugavuse analüüs

Antud bakalarausetöö eesmärkideks on analüüsida populaarseimad soovitusalgoritme, kirjeldada neid lihtsalt loetaval viisil ning analüüsida neid implementeerivaid tarkvara raamistike kasutusmugavust.

Teises peatükis analüüsitakse algoritme, mida kasutatakse soovitussüsteemides kõige enam. Uuritakse nende põhiideid, erinevusi ning algoritmide tüüpe, nende toimivust, skaleerivust, rakendamist ja täpsust. Analüüs on püütud teha hästi loetavaks ning mõistetavaks.

Kolmandas peatükis uuritakse saadavalolevate soovitussüteemide tarkvararaamistike kasutusmugavust läbi lihtsa soovitussüsteemi loomise kaudu nende raamistikke abil. Vaadeldakse dokumentatsiooni kättesaadavust, konfigureerimise ja integreerimise lihtsust, erinevusi, skaleeritavuse tuge ning kasutuse lihtsust.

Lõpuks, kokkuvõtes tehakse järeldusi tehtud analüüsi alusel. Arutletakse millised algoritmid on sobilikud erinevate kasutusjuhtude puhul ning millal on parem võtta kasutusele üht või teist raamistikku soovitussüsteemi loomisel.

Tehtud analüüs peaks olema piisav, et langetada teadlik otsus soovitusalgoritmide ning raamistike valikul erivaldkonna soovitusüsteemi loomiseks.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 34 leheküljel, 4 peatükki, 11 joonist ja 1 tabel.

List of abbreviations and terms

Weighted average	***Average value of items computed by considering relative importance of each value*** Kaaludega varustatud suuruste keskmine
Data mining	***Computational process of discovering patterns in large datasets*** Automaatne protsess kasulike mustrite paljastamiseks suurtest andmehulkadest
Machine learning	***Study and construction of algorithms that can learn from and make predictions on data*** Teadusvaldkond, mille eesmärk on välja töötada empiiriliste andmete põhjal otsuseid ja ennustusi tegevaid algoritme
Cosine	***Ratio of the length of the adjacent side to the length of the hypotenuse*** Lähiskülje ja hüpotenuusi suhe
Data sparsity	***Percentage of empty cells in the table*** Tühjade väljade osa tabelis
Open source software	***Computer software with its source code made available with a license in which the copyright holder provides the rights to study, change, and distribute the software to anyone and for any purpose*** Tarkvara, mille lähtekood ja dokumentatsioon on kõigile kasutajatele ja arendajatele vabalt kättesaadav nii tutvumiseks kui muutmiseks
Framework	***Abstraction, which provides generic functionality helping to simplify application development*** Abstraktsioon, mis pakub geneerilise funktionaalsuse, aidates rakenduste arendamisel
Java	***An object-oriented programming language*** Objektorienteeritud programmeerimiskeel

API	***Application Programming Interface, a set rules for interacting with existing software*** Reeglistik olemasoleva programmiga suhtlemiseks
Node	***Single active electronic device that is attached to a network*** Üksik seade mis on ühenduses osana arvutivõrgust
Fork	***Creation of separate and distinct software based on an existing software package*** Eraldiseisva tarkvara loomine olemasoleva tarkvarapaketi alusel
Interface	***An abstract type in object-oriented programming languages that contains no data or code, but defines behaviors as method signatures for implementation in child classes*** Abstraktne tüüp objekt-orienteeritud programmeerimiskeeltes milles ei ole andmeid ega koodi, aga mis määrab end implemeteerivate klasside kätumist
IDE	***Integrated Development Environment, software that provides comprehensive facilities to computer programmers for software development*** Tarkvara programm, mis pakub programmeerijatele põhjalikke vahendeid tarkvara arendamiseks

Table of contents

List of figures

List of tables

1 Introduction

Nowadays there is a great amount of data being collected on people's behaviour. Every time someone visits a website online, swipes a loyalty card in the shop or pays for purchases with a credit card all these actions are being recorded. For commercial enterprises, there is great potential in using this vast amount of data in order to generate insights that can lead to increased revenue. The companies that have embraced data analysis as cornerstones of their business, such as Google, Amazon and Netflix, have thrived and become household names.

However, not all companies are taking full advantage of such a valuable resource. Some of the most glaring examples that can be observed in Estonia are supermarket chains. Even though some of them have been collecting information about our shopping habits for decades, we as end customers don't see much personal benefit from it. Indeed, some of these chains have recently introduced some services that are based on data analysis and benefit the consumer. However, these are mostly rudimentary and lacking in personalization. One of the possibilities for improving the status quo is introducing a recommender system to the shopping experience. This approach has already been proven in e-commerce to improve the shopping experience for consumers and increase revenue for companies [2]. It is high time the traditional retailers also adopt it.

Unfortunately, the majority of research papers that deal with this subject are extremely technical and hard to read, especially for people that are not well versed in computer programming or higher mathematics. This makes for a high barrier of entry into the field.

In this thesis I will analyse the most widely used algorithms in recommender systems and try to present the findings in an easy to digest manner, in order to provide a solid understanding of available approaches for potential users. I will also try out and analyse the usability in development of some of the software frameworks that implement such algorithms, which would help potential users to make the right decision when choosing what tool to go with.

2 Analysis of popular algorithms used in recommender systems

2.1 User-based collaborative filtering

Collaborative filtering is one of the earliest and most successful approaches used in recommender systems [15].

The basic premise behind collaborative filtering is to collect and analyse large amounts of data on customers' behaviours, activities and preferences in order to determine whether a product would be useful to the target customer based on the opinions and buying habits of other similar customers [9]. Similarity of customers can be judged based on a multitude of factors, chief among them is the tendency to buy or rate different products in an analogous manner [15]. In essence, if there is a customer who has purchased several items and there is another customer who has purchased, say seventy per cent, of the same items, then it is likely that the second customer would be interested in purchasing the remaining thirty per cent of the items purchased by the first customer. An example of user-based collaborative filtering is given in Figure 1.

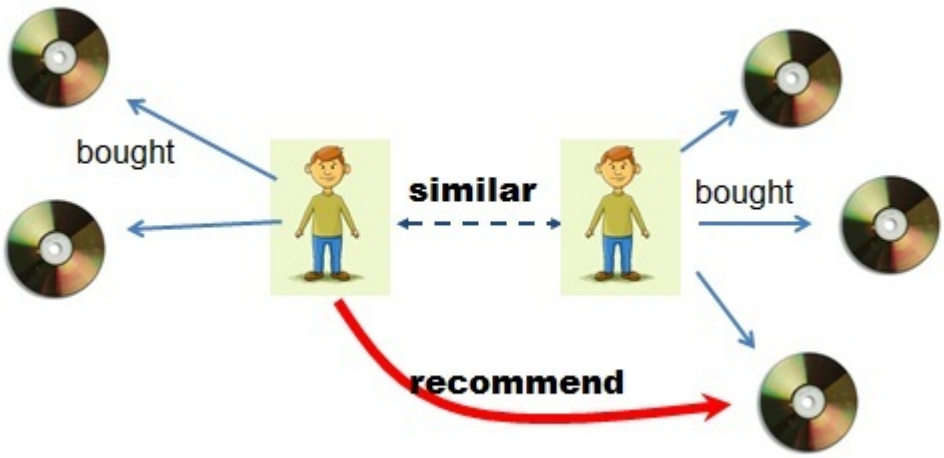

Figure 1. Example of user-based recommendation [3]

2.1.1 Process of user-based collaborative filtering

The process of user-based collaborative filtering can be divided into three main steps - representation, neighbourhood formation and recommendation generation.

Typically, the input data or original representation in recommender systems is the purchasing or rating history of items by customers.

Neighbourhood formation is the most important step in collaborative filtering-based recommender systems. Here, the similarity between the target customer and other compatible customers is calculated. This step is the learning process for the recommender system.

Finally, generation of recommendation is basically deriving the top items bought or rated by the formed neighbourhood of customers. There are three ways of accomplishing this:

- Recommendation of most frequent items - here the purchase history of neighbours (customers) in the neighbourhood is looked over and the frequency count of items is performed. After that process is finished, the items are ranked based on their count and the most popular ones are presented as recommended to the target customer;

- Recommendation based on a rule - this approach takes a predefined rule, usually incorporating a prediction about the result of the computation, and applies it to the formed neighbourhood. For example, the rule can state that if an item was bought together with the items that might be of interest to the target customer, the item itself would also be useful to this customer. Thus this item is recommended;

- Mix of the previous two [15].

The approaches described above are tackled with more detail in the next chapter.

2.1.2 Types of user-based collaborative filtering

2.1.2.1 Memory-based

Memory-based approach to collaborative filtering has its goal the prediction of a customer's purchase based on the dataset of purchases of other existing customers. For example, the probability of a customer buying an item is calculated as an average of some similar customers' rate of buying the same item.

One of the ways it can be implemented is by calculating the similarity between the target customer and one other customer, which is the most similar, and taking the weighted average of the rate of purchased items as an indicator for recommendation.

Another way is finding the k most similar customers and aggregating their purchase histories in order to identify the most likely items that the target customer will like [9].

The advantages of this approach are its relative simplicity, independence of the content of the items to the outcome of the computation and explainability of the results. Disadvantages are decrease of performance in the event of data sparsity, which hinders scalability and adds difficulty in introducing new items to the dataset [13].

2.1.2.2 Model-based

A model is a rule created by applying data mining algorithms and machine learning techniques in order to uncover patterns in the existing datasets [4].

There are a multitude of available models, most of them based on mathematical and statistical algorithms such as clustering, where items are grouped into a set in such a way that members of the set are more similar to each other than to members of other sets, Bayesian networks, where relationships between nodes represent conditional dependencies (cause and effect) and classification [4].

Applying the model-based approach to collaborative filtering gives more control over the resulting recommendations as it provides a way to influence them. For example, instead of looking for customers with similar purchasing behaviour regardless of any external factors, such variables as weather conditions, day of week and others can be introduced as a rule, which might significantly affect the end result.

Advantages of using the model-based approach are better handling of data sparsity, which in turn helps scalability, and potentially higher quality of recommendations, as with thoroughly researched models additional solid data points are introduced.

Main disadvantages are difficulty in building models, potential loss of useful information and occasionally difficulty in explaining resulting recommendations [4].

2.1.3 Performance

A compelling feature of user-based collaborative filtering is that this method does not require understanding of the content of complex items being recommended [3]. Instead simply the commonality is used to determine probable usefulness of the products.

Another advantage of using user-based collaborative filtering-based recommender system is that implicitly all possible statistical factors are taken into consideration [4].

However, even though the given approach is one of the most successful it also has its drawbacks. The main ones are:

- The necessity to collect a sizable amount of data in order to provide worthwhile recommendations;
- Scalability issues past several million customers and items [3];
- Inadaptability to new items;
- Synonyms - items with different names referring to the same core product [13].

Accordingly, there are several ways these limitations are handled. In order to deal with data sparsity a variety of techniques are employed, such as automatic rating of the items based on their contents, which allows to provide a more detailed background on each bought product. One way of addressing scalability issues is to group the customers into clusters and limit the search to the one, which correlates with the target user the most. As for introducing new items into the equation the same approach as for dealing with data sparsity can be utilized [4].

2.2 Item-based collaborative filtering

Another approach to finding products that would be of interest to the target customer is by basing the recommendation algorithm not at finding similar customers and their shopping history, but at the items themselves. With this approach, first the similarities between the various items are determined and then used to identify suitable items to be recommended.

As opposed to traditional collaborative filtering, this approach does not require the computation to determine the so-called neighbourhood of comparable customers, which tends to be one of the most performance-intensive parts of generating the recommendation, so the end result is generally achieved much faster. Item-based recommendation approach uses a precomputed model of items' similarities, which enables it to identify a set of items to be recommended quite quickly.

The premise behind this way of pinpointing items of interest is that the customer is more likely to find appealing the items that are related or share similarities with the products that he or she has already purchased [9]. An example of item-based collaborative filtering is given in Figure 2.

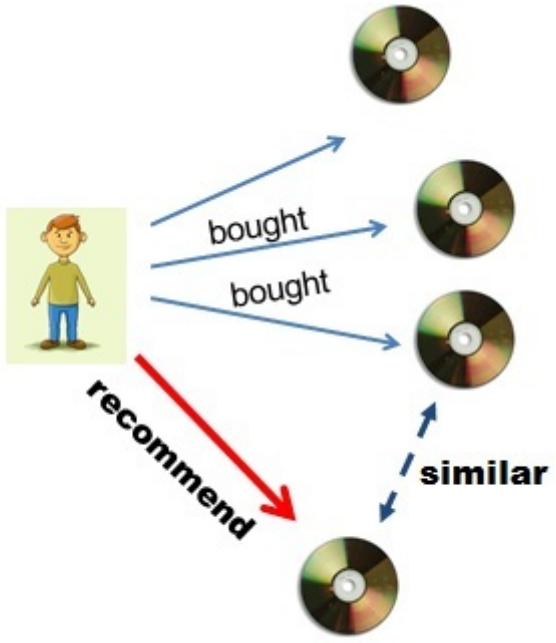

Figure 2. Example of item-based recommendation [3]

2.2.1 Determining item similarity

As mentioned, one of the advantages of item-based approach to finding items of interest is that it requires less real-time overhead in comparison to collaborative filtering, due to the fact that the model on which the recommendation is based is computed in advance of the actual generation of the recommendation. The phase of model building consists of:

- Finding the *n* most similar items for each item;
- Recording the similarities between the items;
- Determining the top recommended items for each customer, based on the set of items from the customer's shopping history.

The last step is performed as follows:

- For each item present in the customer's shopping history the *k* most similar items are found;
- The items that are already present in the customer's shopping history are removed from the resulting set;
- For each item in the remaining set the similarity to the existing shopping history of the customer is computed as the sum of similarities between all the items in the remaining set and all the items in the existing shopping history;
- The items in the resulting set are sorted by similarity rating and the specified number of items is selected as recommended.

```
For each item in product catalogue I1
      For each customer C who purchased I1
            For each item I2 purchased by
                  Customer C
                  Record that a customer purchased I1
                  And I2
      For each item I2
            Compute the similarity between I1 and I2
```

Figure 3. Item-based recommendation algorithm in pseudo code

Example of the described process is given in Figure 3.

The most important part of the described process is determining item similarity. There are two most common ways of doing that [9].

2.2.1.1 Cosine-based similarity

The first way of computing the correlation of items is by treating each item as a vector in customer-space. The similarity is thus represented as a cosine of the angle between such vectors [1].

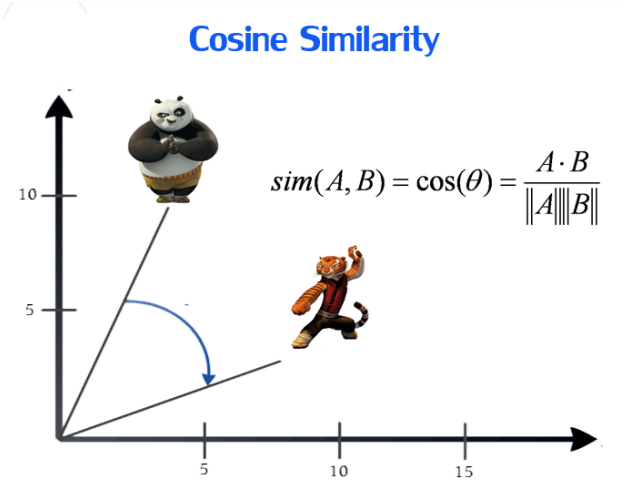

Figure 4. Example of cosine similarity of cartoon characters [3]

As can be seen from Figure 4, if the customer has bought one item and also the other one the similarity between the two items will be rather high. The rate or frequency of purchases is also taken into account, so the items that are bought the most will share a high similarity rating with other items that are bought the most [9].

2.2.1.2 Conditional probability-based similarity

Another way of computing the similarity of items is by measuring the probability of the event that an item is bought, given that a set of other items has already been bought. This can be presented by a simple equation (1).

$$P(u \mid v) = \frac{Freq(uv)}{Freq(v)} \tag{1}$$

What the equation (1) represents is essentially a number of customers that purchase both items v and u divided by the total number of customers that have purchased v.

One of the glaring disadvantages of using such a measure of similarity is that every item in the dataset will have a high similarity rating to the items that are bought the most frequently, caused simply by the fact that popular items are bought more frequently and other items tend to be bought together with them. In order to combat this shortcoming researchers have come up with several strategies the most straightforward of which is simply dividing the resulting probability of similarity by the frequency of occurrence of the each item in the dataset [9].

2.2.1.3 Normalization of similarity

As the frequency of item purchases can vary greatly so can the similarity rating of items. The products that are bought less often can potentially have a big influence on the overall result of determining the suitable recommendations if they have even a moderate correlation with other items, that are also bought infrequently. This can manifest itself in unsuitable recommendations. In order to deal with this shortcoming similarity normalization techniques are used that place more emphasis on less active customers, whose shopping history data is more insightful, and deemphasizes the more frequently purchased items, the data on which can be polluting. Similarity normalization has been demonstrated to bring improvements in quality of recommendations up to the tune of twelve per cent [9].

2.2.2 Performance

Performance-intensiveness of item-based recommendation algorithms is calculated depending on two factors:

- The amount of time it takes to compute the most similar items for each item (creating a model);
- The amount of time it takes to generate recommendations based on the created model.

As a result of the fact that most customers tend to buy a small number of products, which are usually clustered, the data sparsity level can be very high. Consequently, this allows to perform the necessary computations only between pairs of products that have been bought together at least once, removing the need to take into account all the data, that does not conform to this requirement. In some datasets the sparsity can be as high

as ninety nine per cent, which means the recommendation is generated by only looking at one per cent of the whole dataset [9].

The described feature is a big performance advantage in comparison to traditional user-based collaborative filtering, where the whole dataset needs to be combed through in order to generate the recommendation, and is especially useful in real-time systems.

In addition, a big advantage is that the most data-intensive computations, namely looking up similar items for each item, are performed offline. This allows scalability independent of the total number of items in the dataset [12].

Another notable quality of item-based algorithms is their tendency to generate only gradually decreasing quality of recommendations with smaller datasets. Interestingly, studies have shown that by shrinking the dataset fivefold the quality of recommendations only suffered to the tune of less than two per cent [9]. In fact, high-quality recommendations can be achieved with as few as several item pairs in the dataset [12].

Where item-based algorithms suffer is the personalization. Generally, user-based approach where the results are calculated from the data of the neighbourhood of most similar users, tends to offer more personal recommendations [9].

2.3 Hybrid user-based and item-based recommendation algorithms

User-based and items-based approaches to generating recommendations can be combined with the goal of taking advantage of the benefits of both.

The implementation of such algorithm can vary from first identifying a neighbourhood of similar customers and then applying item-based filtering to it [9] to running both algorithms independently and combining the results [3].

There are several studies that compare the performance of the hybrid approach with the strictly collaborative and item-based algorithms. They show that the hybrid algorithms can provide more accurate recommendations than pure approaches. This approach can also be used to overcome some of the common issues in recommender systems such as cold start and the data sparsity problem [9].

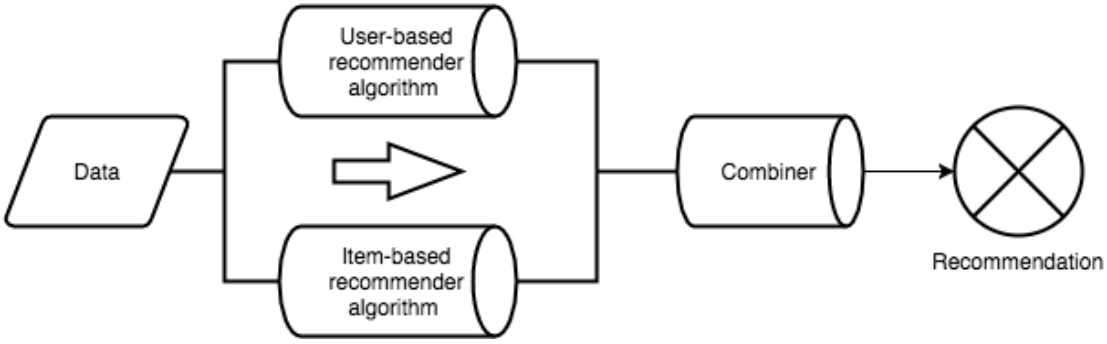

Figure 5. Example of hybrid recommendation algorithm

2.4 Matrix factorization approach

Matrix factorization is an alternate approach to generating recommendations. Such algorithms, instead of finding similarity between customers or items, look for a multitude of factors, called dimensions, gathered from the shopping or rating history patterns. For example, for edible products this can be expiration date at time of sale, seasonal trends, age of customer and others. Another good example of such approach by a prominent user of this algorithm - Netflix - is categorizing movies into dimensions, where a dimension can correspond to a genre, target demographic (male or female) and the general feel of a movie. Figure 6 demonstrates this example.

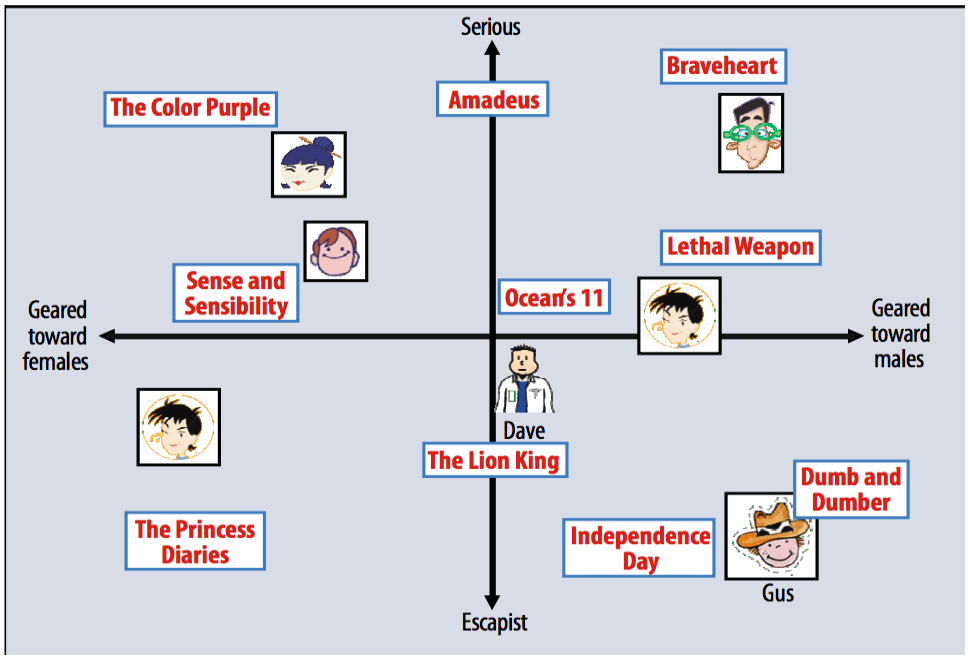

Figure 6. Simplified example of matrix factorization [10]

In matrix factorization interactions (purchase or rating) between a customer and an item are placed onto a single space of factors where each customer and item is associated with a vector, which represents positive or negative extent, to which the customer is interested in the factors or the item possesses the factors, of which the space is comprised. The result of such mapping is a matrix of dots, which captures the customer's interest in any given item. A simplified version of this can again be observed on Figure 6.

The main difficulty in this approach is computing the user-item mapping [7].

2.4.1 Singular value decomposition applied to matrix factorization

Matrix factorization is very similar to a well-known technique for information retrieval called singular value decomposition (SVD). SVD is basically decomposition of a matrix into matrices of lower dimension. These matrices are useful for recommender systems because they allow to reduce computational overhead by producing a low-dimensional representation of the initial space of customers and items, thus enabling calculations on much reduced dataset. They also help with the main goal of matrix factorization by enabling capture of relationships between customers and items, which allows prediction of likelihood of customer's interest in an item [10].

2.4.2 Adding extra inputs to matrix factorization

A great strength of matrix factorization is that it allows for input of additional information to the dataset, when existing data is lacking or needs modifications to comply with business rules [10].

In matrix factorization this additional data is called a bias. Biases can be used to reduce or magnify the weight of necessary factors. One example is a critical customer, who tends to rate an item lower than other customers. In this case such customer's ratings of items can be given a lower weight when looking for items to recommend.

Another example is adding biases based on the customer's behaviour not directly related to any products. In e-commerce this can be browsing and search history, location data and so on. In retail it can be, for example, weather data and distance of customer's home to the retail location. Such biases can greatly alleviate a common problem of recommender systems working with a small dataset - a so-called cold start [10].

2.4.3 Dynamic recommendations

As customer's tastes and products' popularity change over time, recommender systems need to provide up to date suggestions. To this extent matrix factorization offers decomposition of purchase or rating history over different time periods.

The fact that an item's perception can change in time is addressed as item biases, which represent item's popularity as a function of time. Customer's perception of items can also vary, wherein a user might hold a product in high regard at one point and change his or her opinion at another point, perhaps due to peer pressure or new knowledge about an item. For such cases user bias can also be represented as a function of time.

Fleeting trends are also addressed by matrix factorization. It can work with less meaningful data caused by varying interest in products and assign less weight to such trends. One-time events are recognized as such and the recurring events are treated as more indicative of a customer's opinion [10].

2.4.4 Performance

Matrix factorization lends itself well to scalability due to the possibility of working on smaller (decomposed) dataset matrices.

Quality of predictions is dependent on refinement level of factor space, namely the accuracy in choosing and describing the parameters of the space, which can number in the hundreds of millions. Generally, the quality of recommendations is considered to be of very high level.

Nowadays matrix factorization has become the dominant approach to recommender systems [10].

3 Analysis of recommender system frameworks' usability

With the rise in popularity of applying recommendation algorithms to real-world data, there have emerged several software engineering community-supported frameworks, which assist the developers with creating software that utilizes the mentioned algorithms to help the end users with finding interesting products. As Java is the most popular programming language [16], the frameworks based on it were chosen for evaluation. Two frameworks were chosen for evaluation because of their availability in the Maven Central repository and richness of documentation. These were Apache Mahout and LensKit. Another one - LibRec - was chosen due to offering many modern algorithms, that the first two do not provide out of the box [6]. These frameworks offer essentially the same base algorithms. They, however, differ in implementation, data management and evaluation ratings, all of which has an effect on usability, performance and results. There exists a multitude of performance and results-oriented publications about these frameworks [6],[14]. However, they overlook an important side of the frameworks which is ease of use. In the following chapters this topic will be examined more closely by building a simple recommender engine using each of theme and evaluating the difficulty of doing so. The data for testing the systems is a small dataset crawled from the Filmtrust website containing 35497 item ratings which is provided by the creators of LibRec [(A Collection of Recommendation Data Sets, 2011)].

3.1 Apache Mahout

Apache Mahout is an open source Java framework that provides an API for building applications, which require use of scalable machine learning algorithms. It is licensed under the Apache License, meaning it is free to use, modify and distribute. At the time of writing the latest version of the framework is 0.12, but despite the major version number being zero it is quite mature. Some of the main goals of the Mahout project are focusing of practical use cases, instead of unproven techniques or new and raw research and providing quality documentation, which, together with being free to use, makes it an attractive candidate for use in commercial systems [7].

One of the prominent uses cases for Mahout is in recommender systems, as it offers via its API implementations of many algorithms required in such systems. These include all of the ones that were analysed in the first part of this thesis - user based collaborative filtering, item based collaborative filtering and matrix factorization. It has to be noted that some researchers have pointed out that Mahout focuses on memory-based algorithms and might be becoming out-dated [6].

Apache Mahout's core algorithms are implemented in two modes: one for use on a single node, called Taste, and another one for use in distributed multi node systems, which are necessary for maintaining performance when working with large datasets. For simplicity's sake we will focus on the single node mode when evaluating ease of creating a recommender system from scratch.

3.1.1 Usability in development

As Mahout is a Java-based framework the prerequisite to using it is having Java installed on the machine. The latest Mahout version, which is 0.12 at the time of writing requires Java version no lower than 7. Installing Java is very straightforward and the developers, who work in it ecosystem usually already have it available on their machines. Being a Java-based library, Mahout is cross platform, meaning it can run on any device supporting Java.

Apache Mahout is available as a standalone Java library, which can be added to any Java application just like any other library. A very convenient feature of Mahout is its availability in the most widely used library repositories, such as Maven Central. This allows for declaring the library as a dependency for the automated application build tools like Maven or Gradle, that are used in the vast majority of software projects [8] in order to have a automated, structured and maintainable process on including external tools used in development. In practice this means that integrating Mahout into an application is as easy as declaring it a dependency in the project build file, as demonstrated in Figure 7.

```
<dependencies>
    <dependency>
        <groupId>org.apache.mahout</groupId>
        <artifactId>mahout-mr</artifactId>
        <version>0.12.0</version>
    </dependency>
</dependencies>
```

Figure 7. Example of Mahout dependency declaration in Maven

When Mahout is integrated into the project and ready to be used, there are several relatively simple steps needed to make it generate recommendations:

- Preparing data in the format userId, itemId, rating (can be any numeric value, the higher the better rating);
- Preparing the model (tuning the algorithm);
- Integrating with the application that will use the recommendations.

The data can be fetched from either the database, by implementing the interface `JDBCDataModel` offering database-agnostic access to the data store, or a file with comma-separated data values, using the existing class `FileDataModel`. Once that is done, an implementation of the interface `UserSimilarity` is used to define the similarity between users on a scale of -1.0 to 1, where 1 represents perfect similarity. Mahout provides out of the box multiple implementations for `UserSimilarity`, covering the major algorithms used for determining similar customers, such as Pearson correlation and Euclidian distance. Then, in case of user-based recommendation, the neighbourhood of the most similar users is determined by using an implementation of the `UserNeighborhood` interface based on previously defined `UserSimilarity`. In case of item-based recommendation an implementation of the interface named `ItemSimilarity` needs to be used, which determines similarity the same way as `UserSimilarity`, but between items. Finally, in order to generate the actual recommendation, a function called `recommend`, taking as parameters one of the previously defined similarities and a userId, from an implementation of `UserBasedRecommender` or `ItemBasedRecommender` is used. The collection that is returned from the function contains objects of type `RecommendedItem`, which store itemId and recommendation score for the item.

```
public List<RecommendedItem> recommend(long userId, int
numberOfItemsToRecommend) throws IOException, TasteException {
    DataModel model = new FileDataModel(new
File("/Users/ivan.studenikin/ratings.csv"));
    UserSimilarity userSimilarity = new PearsonCorrelationSimilarity(model);
    int neighborhoodSize = 10;
    UserNeighborhood neighborhood = new
NearestNUserNeighborhood(neighborhoodSize, userSimilarity, model);
    UserBasedRecommender userBasedRecommender = new
GenericUserBasedRecommender(model, neighborhood, userSimilarity);
    return userBasedRecommender.recommend(userId, numberOfItemsToRecommend);
}
```

Figure 8. Example of a simple fully functional user-based recommender

In case of using matrix factorization the steps required are the same, but instead an implementation of the `Factorizer` interface and one of the applicable `Recommender` implementations is used.

```
public List<RecommendedItem> recommendSVD(long userId, int
numberOfItemsToRecommend) throws IOException, TasteException {
    DataModel model = new FileDataModel(new
File("/Users/ivan.studenikin/ratings.csv"));
    int numberOfFeatures = 5;
    int numberOfIterations = 100;
    Factorizer factorizer = new SVDPlusPlusFactorizer(model,
numberOfFeatures, numberOfIterations);
    Recommender recommender = new SVDRecommender(model, factorizer);
    return recommender.recommend(userId, numberOfItemsToRecommend);
}
```

Figure 9. Example of a simple fully functional matrix factorization-based recommender

As can be seen from Figures 8 and 9 the interfaces, their implementations and functions have self-explanatory names, which makes Mahout's API quite pleasant to use. It can also be noted that the amount of code required to build a fully functional recommender system is very small. A lot of popular algorithms are already implemented within the API. However, new ones can easily be added or the existing ones customized to better infer business rules.

Mahout also provides a web interface to expose the recommendation results via the web or internal network out of the box. This makes integration with external systems easier. Documentation is quite extensive and covers a lot of use cases. Examples are also provided.

3.2 LibRec

LibRec is an open source library used in recommender systems, offering implementations of a large suite of recommendation algorithms. It is licensed under GNU General Public License, meaning it is free to use, share and copy. LibRec is touted as a much faster alternative to other libraries [6], however, verification of this claim is out of scope of this thesis. It has to be noted though that due to modern algorithms doing the most performance-intensive computations offline and various optimization techniques in use, this advantage has limited it current use cases.

The latest LibRec version, which is 1.3 at the time of writing requires Java version no lower than 7, so there's no difference in comparison to Apache Mahout. LibRec is also cross platform, being a Java-based library.

LibRec offers recommendation algorithms based on work of researchers and each available one has a reference to paper that it was taken from in the documentation. This approach means that a lot of very modern and not widely used algorithms are offered.

There is no mention of support for scalability in the documentation, so an assumption can be made that this is not available out of the box. However, this also means that whoever uses the library is free to choose any suitable technology to add this functionality, if the requirement for making the recommendation system work with massive datasets arises.

3.2.1 Usability in development

Unfortunately, at the time of writing LibRec was not present in any major library repositories, making its use in development of commercial systems more cumbersome. This means that the library and its dependencies have to be added and maintained in a project by hand.

Once the library and its dependencies were downloaded and added to the application, an attempt was made to run an example provided in the documentation. This attempt failed due to the main function used for generating recommendations having protected access, meaning that any applications that are to use LibRec as a dependency need to be essentially a fork of LibRec itself. This may be a disadvantage, that commercial users will not look past. Although LibRec, just like Mahout, also offers a command line

interface that could be used to mitigate the problem, its use is explicitly discouraged in the documentation, due to, again, having to deal with external dependencies by hand.

Having checked out and imported into the IDE the source code of the library, the next step was to configure the application to generate recommendations. In LibRec this is done mostly via a single configuration file. For some developers having to configure the system not programmatically, but via a separate configuration file might be a turnoff, as it was for the author of the thesis. In contrast to Mahout, the naming pattern is lacking in clarity, as demonstrated in Appendix 2. It takes quite a lot of time to understand the meaning of the majority of parameters and their values. It has to be noted though, that all of the options have a description in the documentation.

Another downside of LibRec is that it currently only works with files, containing comma-separated values, so no database connectivity is provided, which is quite inconvenient for development, as purchasing and rating data is usually stored in the database. A workaround for copying the necessary values from the database to a file is required to mitigate this issue. As the data needs to be kept up to date, this might become a rather cumbersome step in the application's lifecycle.

After having setup the configuration and the data, creating the code to actually run the application is quite simple: basically, only the location of the configuration file needs to be specified, as can be observed from Figure 10.

```
public void recommend(String[] args) throws Exception {
    String configFile = "librec/src/main/resources/librec.conf";
    LibRec librec = new LibRec();
    librec.setConfigFiles(configFile);
    librec.execute(args);
}
```

Figure 10. Example of a function required to generate recommendations using LibRec

One obvious downside here is that the function called `execute` does not return any value, but instead either outputs the result to a file or clipboard, based on the configuration setting. This means that in order to usable in commercial systems, a workaround has to be created to allow further usage of the results of the computation in the application. No web interface is provided though, so integration with external systems will have to be done by hand.

The documentation for LibRec covers all of the algorithms offered, explains the meaning and use of the configuration parameters, and offers examples on how to create an application using the library, so the information is quite extensive. Another useful thing offered is several datasets, which can be used to test the performance of different algorithms.

3.3 LensKit

LensKit is also an open source framework for recommender systems available under the GNU Lesser General Public License, meaning it is free to use, modify and integrate, but not or proprietary components. It offers implementation of four recommender algorithms: user-based collaborative filtering, item-based collaborative filtering, matrix factorization and Slope-One.

The latest LensKit version, which is 2.2.1 at the time of writing requires Java version no lower than 6, so it can be said that it is more backwards compatible in terms of Java version that the previous two frameworks [11].

LensKit is touted as useful in research of recommender systems and multiple papers are cited as using it [11].

There is no mention of support for scalability in the documentation, so here also an assumption can be made that this is not available out of the box.

3.3.1 Usability in development

A convenient feature of LensKit is that it is available in major public repositories for dependencies, such as Maven Central. It makes integrating it into an existing application rather fast.

After LensKit is integrated similar steps to the previously discussed frameworks are to be performed: data grooming into the format "userId, itemId, ranking", algorithm configuration and integration with existing system, if applicable. The configuration is done in the source code via the library's API, which is quite convenient. At first, an `EventDao` needs to be set to define the data source. It can be either the database or a file with comma-separated values. Support for both is provided out of the box. Then `LenskitConfiguration` is created and `EventDao` is added to it. Afterwards we

need to bind algorithm specific `ItemScorer` to the configuration. When configuration is finished, we create a `Recommender` that takes the previously defined `LenskitConfiguration` as a parameter. The recommender's method `recommend` then takes the userId and number of items to recommend as parameters and returns a collection of recommendations of type `ScoredId`, which contains within itself the itemId and score of recommendation that can be used further in the application. The basic configuration seems quite easy to setup, although the amount of code required to create a simple recommender is more that Mahout or LibRec need. This is clear from Figure 11.

```
public List<ScoredId> recommend() {
    LenskitConfiguration config = new LenskitConfiguration();
    config.addComponent(new SimpleFileRatingDAO(inputFile, ","));
    config.bind(ItemScorer.class).to(FunkSVDItemScorer.class);
    config.bind(BaselineScorer.class,
ItemScorer.class).to(UserMeanItemScorer.class);
    config.bind(UserMeanBaseline.class,
ItemScorer.class).to(ItemMeanRatingItemScorer.class);
    Recommender rec;
    try {
        rec = LenskitRecommender.build(config);
    } catch (RecommenderBuildException e) {
        throw new RuntimeException("recommender build failed", e);
    }
    ItemRecommender itemRecommender = rec.getItemRecommender();
    return itemRecommender.recommend(111, 10);
}
```

Figure 11. Example of LensKit configuration

The library provides no interface to external systems, so it will have to be implemented by hand, if required. The documentation for LensKit is comparable in size to Mahout, there are clear explanations as to which component is responsible for what and extensive examples are provided.

3.4 Comparison of frameworks' performance

In this chapter I will present and analyse results of the computations performed by the simple recommender systems that were created in the previous chapters. In order to harmonise the comparison the Filmtrust dataset and matrix factorization algorithm will be used in all cases. Algorithm specific settings such as number of iterations (100) and features (40) will also be the same for all frameworks. Despite the fact that the main settings are the same, some of the additional ones are framework specific due to inability to override them. The recommendations were generated for a single user with

userId 111 and were computed five times, with the average values taken into consideration. The findings are presented in Table 1 in the format itemId/score. It has to be noted that the score does not hold any semantic value, but rather is just a way to express preference in the hierarchy of suggestions. The higher the score, the more likely the recommendation will be of high quality.

Apache Mahout (took 29299 ms on average)	LibRec (took 21044 ms on average)	LensKit (took 4391 ms on average)
312/3.60	318/3.98	68/3.53
309/3.48	309/3.44	312/3.46
1517/3.44	432/3.30	1179/3.44
187/3.35	17/3.21	1167/3.44
463/3.23	400/3.10	1865/3.44
338/3.20	248/3.03	1866/3.44
476/3.20	434/3.01	867/3.43
1537/3.18	1091/3.00	854/3.43
261/3.17	1167/3.00	894/3.43
1443/3.16	1972/2.94	162/3.43

Table 1. Recommendations and their scores generated by the frameworks

What is obvious from Table 1 is that the recommended items only correlate once, meaning that in order to generate relevant recommendations probably a fair amount of domain specific fine-tuning of the frameworks is required and out of the box functionality is not enough to do that. However, we can also note that the scores are very similar in terms of numeric value so the scale at which the basic algorithm operates is comparable. What can also be seen is that LensKit is on average quite a bit faster than the other two. This however might be due to a rather small dataset, as studies have shown that LibRec might actually be the fastest [6].

The source code used in the comparison is available on GitHub [5]. Further evaluation and tuning of the frameworks is ongoing, as more up to date datasets become available via collaboration with local companies.

4 Summary

The main goals of this thesis were analysis of popular algorithms used in recommender systems, their presentation in an easy to digest manner and analysis of usability of software frameworks that implement these algorithms.

In the thesis I tried to explain ideas behind different approaches in an easily readable way.

The algorithms presented in this thesis all have their use cases, so a selection of a particular one should be based on the requirements of a concrete system. If no in-depth classification of the products is a feature of the system or there is no need for high performance, then the use of user-based collaborative filtering might be the optimal solution. If performance is of great importance, such as in real time systems, then item-based collaborative filtering can be used. If inclusions of additional factors into the recommender system or dynamic time- or trend-based recommendations are required, the matrix factorization is the better approach. In addition, all the algorithms can be combined. The main correlation between all three analysed algorithms is that the more data is available to be processed, the better the quality of recommendations will be.

As for the usability of available software frameworks implementing the algorithms, it seems that Apache Mahout is the more suitable choice for building commercial-grade recommender systems, as it is very easy to use and integrate into the project, offers support for scalable systems and the most popular algorithms. However, if a system is to be highly innovative and requires usage of multiple bleeding edge algorithms, then LibRec is a better choice, despite the need to highly customize the library. LensKit, on the other hand, might be useful as a simpler and faster alternative to Mahout for performing research on recommender systems.

References

[1] Almazro, D., Shahatah, G., Albdulkarim, L., Kherees, M., Martinez, R., & Nzoukou, W. (2010). A Survey Paper on Recommender Systems. [*Online*] ResearchGate (15.04.2016)

[2] Amazon's recommendation secret. (2012). [WWW] http://fortune.com/2012/07/30/amazons-recommendation-secret/ (15.05.2016)

[3] An Introduction to Recommendation Engines. (2015). [WWW] http://dataconomy.com/an-introduction-to-recommendation-engines/ (13.05.2016)

[4] Breese, J., Heckerman, D., & Kadie, C. (1998). Empirical Analysis of Predictive Algorithms for Collaborative Filtering. [*Online*] Microsoft Research (16.04.2016)

[5] Github: ivanstudenikin/recommenders. (2016). [WWW] https://github.com/ivanstudenikin/recommenders (23.05.2016)

[6] Guo, G., Zhang, J., Sun, Z., & Yorke-Smith, N. (2015). LibRec: A Java Library for Recommender Systems. - UMAP-ExtProc 2015, Dublin, Ireland, June 29-July 3, 2015: UMAP 2015 Extended Proceedings. [*Online*] CEUR-WS (02.05.2015)

[7] Introducing Apache Mahout. (2009). [WWW] http://www.ibm.com/developerworks/java/library/j-mahout/ (05.05.2016)

[8] Java Build Tools. (2016). [WWW] http://zeroturnaround.com/rebellabs/java-build-tools-part-1-an-introductory-crash-course-to-getting-started-with-maven-gradle-and-ant-ivy/ (17.06.2016)

[9] Karypis, G. (2000). Evaluation of Item-Based Top-N Recommendation Algorithms. - Proceedings of the tenth international conference on information and knowledge management, New York, NY, USA : ACM, 247-254.

[10] Koren, Y., Bell, R., & Volinsky, C. (2009). Matrix Factorization Techniques for Recommender Systems. [*Online*] IEEE Computer Society (06.05.2016)

[11] LensKit documentation. (2016). [WWW] http://lenskit.org/documentation/ (17.06.2016)

[12] Linden, G., Smith, B., & York, J. (2003). Amazon.com: Item-to-Item Collaborative Filtering. [*Online*] IEEE Computer Society (06.05.2016)

[13] Recommendation Engine. (2016). [WWW] http://auguricorp.com/auguri/recommendation-engine (30.04.2016)

[14] Said, A., & Bellogin, A. (2014). Comparative Recommender System Evaluation: Benchmarking Recommendation Frameworks. - Proceedings of the 8th international conference on information and knowledge management, New York, NY, USA : ACM, 129-136.

[15] Sarwar, B., Karypis, G., Konstan, J., & Riedl, J. (2000). Analysis of Recommendation Algorithms for E-Commerce. - Proceedings of the 2nd ACM conference on electronic commerce, New York, NY, USA : ACM, 158-167.

[16] TIOBE Index for May 2016. (2016).[WWW] http://www.tiobe.com/tiobe_index (13.05.2016)

Appendix 1 - Comparison of usability of recommender frameworks

	Apache Mahout	LibRec	LensKit
Open source	yes	yes	yes
Available on Maven Central	yes	no	yes
Extensible	yes	yes	yes
Requires forking	no	yes	no
Supports scalability	yes	no	no
Offers web interface	yes	no	no
Self explanatory API	yes	no	yes
Offers database interface	yes	no	yes

Appendix 2 - Example of LibRec configuration file

```
dataset.ratings.lins=/Users/ivan.studenikin/ratings.txt
dataset.social.lins=-1
ratings.setup=-columns 0 1 2 -threshold -1 --time-unit SECONDS
recommender=PMF
evaluation.setup=cv -k 5 --test-view all --early-stop RMSE
item.ranking=off -topN -1 -ignore -1
output.setup=on -dir ./Results/ -verbose on, off --to-clipboard
guava.cache.spec=maximumSize=200,expireAfterAccess=2m
num.factors=10
num.max.iter=100
learn.rate=0.001 -max -1 -bold-driver
reg.lambda=0.1 -u 0.001 -i 0.001 -b 0.001 -s 0.001
pgm.setup=-alpha 2 -beta 0.5 -burn-in 300 -sample-lag 10 -interval 100
similarity=PCC
num.shrinkage=-1
num.neighbors=50
AoBPR=-lambda 0.3
BUCM=-gamma 0.5
BHfree=-k 10 -l 10 -gamma 0.2 -sigma 0.01
FISM=-rho 100 -alpha 0.5
GBPR=-rho 0.8 -gSize 5
GPLSA=-q 5 -b 0.4
Hybrid=-lambda 0.5
LDCC=-ku 20 -kv 19 -au 1 -av 1 -beta 1
PD=-sigma 2.5
PRankD=-alpha 20
RankALS=-sw on
RSTE=-alpha 0.4
SLIM=-l1 1 -l2 5 -k 50
SoRec=-c 1 -z 0.001
SoReg=-beta 0.01
timeSVD++=-beta 0.4 -bins 30
TrustMF=-m T
WRMF=-alpha 1
```