Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Laki Dániel

EFFICIENT SESSION-BASED ITEM-TO-ITEM RECOMMENDATION

DEPARTMENTAL CONSULTANT

Simon Gábor

EXTERNAL CONSULTANT

Daróczy Bálint

BUDAPEST, 2016

Table of contents

HALLGATÓI NYILATKOZAT

Alulírott **Laki Dániel**, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2016. 12. 17.

...…….
Laki Dániel

Összefoglaló

Az ajánlórendszerek mint kutatási terület több új iránya is az elmúlt években keletkezett. Explicit ajánlóknál [1][2] a legelterjedtebb mérték mai napig az RMSE (Root Mean Squared Error), melynek legnagyobb hibája, hogy nem veszi figyelembe az elemek sorrendjét. Utóbbi a valós felhasználás során rendkívül fontos, hiszen a gyakorlatban a felhasználók számára csak egy rövid, de reményeink szerint releváns lista megmutatására van csak lehetőség.

A probléma felismerése után nem csak új algoritmusok keletkeztek [3], hanem felmerült többek között a rangsor alapú kiértékelés, mint pl. az nDCG (normalized Discounted Cumulative Gain) illetve kontextuális elemek felhasználása [4][5]. A hagyományos CF (Collaborative Filtering) ajánlórendszerek esetén feltételezés, hogy a felhasználó már értékelt vagy legalább megtekintett több elemet a rendszerben. Eme feltételezés nem minden esetben igaz. Gyakran előfordul, hogy egy honlap látogatói „véletlen sétákat” alkotva (session), bejelentkezés nélkül keresik a megfelelő elemet [6][7].

Ezen dolgozat célja egy olyan ajánlórendszer fejlesztése, amely képes ajánlásokat adni bármely látogató számára. Ebbe beletartoznak az aktív felhasználóknak szóló testreszabottabb, és az első látogatók számára szóló általánosabb ajánlások is. A felhasználóknak szóló ajánlásokon túl a rendszernek képesnek kell lennie kifejezetten a tartalmakhoz köthető ajánlások adására is.

A dolgozat több ajánlómodell bemutatását is tartalmazza, melyek a terület legfrissebb kutatási eredményei alapján kerültek kiválasztásra és implementálásra. Ezek a modellek számos szimuláció során kerülnek összehasonlításra, az iparban elterjedt kiértékelési módszereket alkalmazva. Ezen szimulációk azt is vizsgálják, hogy különböző frissítési stratégiák milyen hatással vannak a modellek teljesítményére.

Abstract

Many new areas of research rose in the past few years in the field of recommendation systems. Still to this day, the most widespread measure for explicit recommenders [1][2] is RMSE (Root Mean Squared Error). Its biggest mistake is disregarding the order of elements, which has a great importance in real usage, since in practice, it is only possible the show the users a short, but hopefully relevant list.

After realizing this problem, not only new algorithms were created [3], but ranking-based evaluation (e.g.: nDCG (normalized Discounted Cumulative Gain)), and the usage of contextual elements were also considered [4][5]. In the case of traditional CF (Collaborative Filtering) recommendation systems, it is assumed that the user already rated, or at least viewed multiple elements in the system. This assumption is not always true. As it is often the case, the visitors of a website may be looking for certain elements without logging in, going through "random paths" (session) [6][7].

The goal of this thesis is to develop a recommendation system, that can provide recommendations to any user visiting the site. This includes both highly personalized recommendations for already existing active users, and more general recommendations for first time visitors. It also includes giving recommendations for both users, and for items.

The thesis contains an introduction of multiple recommendation models based on the latest results in the area. These models are implemented, and then compared through several simulations, evaluated with measurements that are widely used in the industry. Also examined are the effects of differently scheduled updates on these models.

1 Introduction

Nowadays, personalized experience is everywhere. Websites try to show content to the visitors, that might be most relevant to their interests. Content providers constantly try to show new content to the users, that they might be most interested in. Online stores try to recommend items to returning visitors based on their history. These are all done using different recommendation systems. The quality of the results these systems can give is critical. So much, that in 2009 Netflix offered a prize of $1,000,000 for the team that can make the biggest improvement on their recommendation algorithm [8].

In recent years, recommender systems have been a very active area of research. Many of the previously widespread algorithms (such as k-Nearest-Neighbors) were found to be relatively ineffective compared to ones using Matrix Factorization. Because of this, in the first part of the thesis, an introduction is given to two variations of Matrix Factorization models that I implemented. I also introduce the previously mentioned k-Nearest-Neighbors, that I implemented as well, to serve as a basis for comparison. This chapter also includes an introduction to different evaluation methods, that are widely used in the industry.

The mentioned algorithms cover giving recommendations to a user based on his/her history. Another approach however is item-to-item recommendation. In such systems, users and their actions are not the basis for recommendations. Instead, items are recommended based on how similar they are to other items. This is something that can often be seen in web shops: when a user visits an item, he can see other items, that are similar to the currently viewed item. The second part of the thesis introduces two models, that work this way. One of them is a rather straightforward, and resource-friendly algorithm. The other one [7] is a more complex, supposedly state-of-the-art algorithm.

One thing that should be considered when designing a recommender system, is the update schedule. Well-scheduled updates can mean the difference between a very good, and an unusable recommender. While the model might work well for a while after the initial training, this is bound to change. More users join the system, more items are added, and the users' tastes change over time. Because of this, frequent updates are necessary. On the other hand, too frequent updates can also ruin the system. Running an update is usually very resource-consuming. While some models can do it in seconds, there

are models that might require hours, or even days to run an update. For this reason, an update should only be executed when the models' quality is known to be decreasing. Because of this, I created an Update Proposer, that keeps track of the model it is watching, and decides when an update is necessary. That proposer is introduced in the next chapter.

A recommender algorithm is only truly useful, when it is part of a bigger system. In this case, the recommender system was created to provide a larger application with recommendations. To understand the conditions better, the next chapter introduces architecture of the recommender framework, and the application it is used in.

The second half of the thesis mostly consists of the conditions and results of various simulations. The performance of each model is evaluated, and compared on different datasets. After a meta-model is proposed to improve the results, further comparisons are made between the performance of the meta-model, and the original models. Different scenarios are compared for scheduling updates. Finally, the system-wide performance issues are addressed.

2 User-to-item Models

This section contains a description of three well-known models that are often used for recommendation systems. The first two models are *Matrix Factorization* models. The first one uses *Stochastic Gradient Descent*, while the second one uses *Alternating Least Squares*. The third model is *k-Nearest-Neighbors*. All three models can have different parameters, the ones that were used here can be found in the appendix.

2.1 Matrix Factorization with Stochastic Gradient Descent

The following section introduces the first model, *Matrix Factorization* with *Stochastic Gradient Descent* (or *SGD*). The general idea is to construct a matrix, that contains all the known values (in this case, ratings) for user-item pairs. The goal is to predict the unknown values based on this matrix.

2.1.1 Preprocessing

Theoretically, the matrix constructed in the first step could already be used to start the factorization. However, results can be improved a lot by using some additional techniques beforehand.

Perhaps one of the most important of these techniques is normalization, or centralization. Some users like to give generally higher ratings to everything. This means that even though two users' ratings can be entirely different, their relative ratings might be the same. Because of that, it is recommended to subtract the average of a user's ratings from his ratings, and perform the factorization on the resulting matrix. The same can be done with items too on the resulting matrix. Using centralization alone can already give a somewhat decent prediction to the unknown elements.

2.1.2 Matrix factorization

The idea behind matrix factorization is the following: instead of working with a matrix with a very large number of elements, it can be easier to work with multiple smaller matrices that can produce the original matrix. While it is also possible to drastically reduce memory-usage this way, this is not the only reason it is popular in recommender systems.

The matrix containing the known user-item pairs is extremely sparse. Realistically, a user only ever sees a very small subset of all the items that the model contains. Because of this, the smaller matrices can only be constructed based on those few items that are known. The assumption is that when these matrices are produced, the result will not only be close to the already known user-item pair, but will also give a good prediction for the values that are not known. In this case, the matrix containing the known values (*M*) will be decomposed into two factor matrices, *U* and *V*. These will produce the prediction matrix (*P*).

The algorithm looks like this:

1. Initialize *U* and *V*
2. Repeat *n* times:
    a. Pick a known value from *M* (*i*,*j*)
    b. Improve *U* and *V* in a way, that the difference between *P[i,j]* and *M[i,j]* decreases.

Each of these steps can have different approaches, the following sections contain a description of the methods used in this model.

2.1.3 Initialization

There are two main factors to consider during the initialization of *U* and *V*. The initial values in *P* should be relatively close to the values in *M*. *P* containing the average of the values in *M* in every field might look like a good choice, except the initial values in *P* should also be somewhat random – measurements show that introducing randomness to the initial matrix can lead to far better results. This leads to the following algorithm:

Let *a* be the "central value". This is the value that disregards any random factor. If all the values in *U* and *V* are set as *a*, then the resulting *P* will contain the average of the values in *M* as every element. Furthermore, let *c* be the maximum distance from *a*. This means that the values in *U* and *V* will be chosen from the range [(*a* - *c*); (*a* + *c*)].

The first step is calculating *a*. Let *d* be the common dimension of *U* and *V*. Then, each element of *P* is:

$$avg(M) = a^2 * d$$

Where *avg(M)* is the average of the **known** elements of M. From this, *a* can be calculated:

$$a = sgn\left(\frac{avg(M)}{d}\right) * \sqrt{\left|\frac{avg(M)}{d}\right|}$$

After that, filling up *U* and *V* is done the following way: just let *r* be a random number in the range [(*a* - *c*); (*a* + *c*)]. Each element of *U* and *V* shall be a new *r*.

2.1.4 Choosing the next value

The next issue is the order, in which the known elements of *M* should be iterated. As with the previous step, the difference is easily measurable between a linear iteration, and a random permutation. Furthermore, it is also recommended to visit the known elements multiple times. So, to determine the iteration order, first a list should be created of all the known elements of *M*. Shuffling this list gives an iteration order. Iteration through the list can be repeated as many times as desired.

2.1.5 Measurements

The real challenge lies in determining the new values of *U* and *V*. The aim is of course to *improve* the resulting matrix (*P*). After the improvement of *U* and *V*, it should contain *better* values, than before. This raises the question: what does *improve* and *better* mean? How can one measure how *good* a *P* matrix is? If the means to determine this value is not known, there is no sense in talking about improving it. Fortunately, multiple measurements like this exist.

2.1.5.1 RMSE

RMSE, or *Root-Mean-Square Error* is a measurement commonly used in recommendation systems. For example, the goal of the 1,000,000$ Netflix challenge was to beat their algorithm's (*CineMatch*) RMSE by 10%.

RMSE gives information about how far the known elements of *M* are from the same elements in the prediction matrix (*P*). The formula is the following:

$$RMSE = \sqrt{\frac{\sum_{(i,j)\in M}(M[i,j] - P[i,j])^2}{|M|}}$$

RMSE is far from ideal. It completely disregards the order of elements, which is critical in recommendation systems. However, it can serve as a very good basis for improving the prediction matrix, and will be used in the algorithm. As for the order, there is a different measurement for that.

2.1.5.2 nDCG

The goal would be to recommend items to the user, which he or she might be the most interested in. It is safe to assume that these are the items that would get the highest ratings from the user in question. Giving the user the top 10 recommended items this way is easy: just sort the list of items of the user, based on each item's predicted rating value, then present the user with the first 10 items in that list. But how good is this list? *nDCG* can give a measurement to that.

nDCG, or *normalized Discounted Cumulative Gain* measures the ranking quality of an algorithm. The basis for ranking is the *relevance* of an item. In this case, this relevance is the rating value belonging to each item. To understand how this works, break it down to smaller parts.

Cumulative Gain is the sum of the relevance of the items:

$$CG_p = \sum_{i=1}^{p} rel_i$$

Where p is the number of items in consideration (e.g. when recommending the best 10 items: $p = 10$). This isn't a very good measure, since it doesn't take into consideration the actual position of each item on the list.

The idea behind **Discounted Cumulative Gain** is that highly relevant items should appear earlier on the list than less relevant, or irrelevant items. Because of that, the considered relevance of an item appearing lower on the list should be penalized. As shown by Wang et al. [9], penalizing logarithmically is a good choice here:

$$DCG_p = rel_1 + \sum_{i=2}^{p} \frac{rel_i}{log_2(i)}$$

Calculating DCG is the most difficult part, but to make this measurement even more useful, one more step is remaining: how does the ranking's DCG compare to the ideal ranking's DCG? This is what **normalized Discounted Cumulative Gain** gives the

answer to. Note, that for calculating nDCG (just like with RMSE), only the known values should be considered. An *ideal ranking* should be created for this calculation. It is the ranking of items based on their known rating values: in case the algorithm is perfect, the predicted ranking should be the same as the ideal ranking.

This gives the following formula:

$$nDCG_p = \frac{DCG_p}{IDCG_p}$$

Where IDCG is the DCG belonging to the ideal ranking.

2.1.5.3 Recall

Recall answers the question "Out of the items the user is interested in, how many did the system manage to recommend?".

$$recall = \frac{number\ of\ recommended\ relevant\ items}{number\ of\ relevant\ items}$$

The recommended items are determined by different "cuts", depending on how many elements the recommender returns. For a top *n* recommender, an item is recommended, if it is ranked in the first *n* items for the user. Recall can be an extremely useful measurement, since the recommender system returns only a small set of items for each user. The quality of this set is extremely important.

2.1.5.4 MPR

MPR, or *Mean Percentile Ranking* is another ranking-based measurement. However, unlike DCG, *MPR* also takes the popularity of each item into consideration. It gives a percentage based on how many items have a better score (in this case, prediction), than the item under inspection, how many have the same score, and how popular each item is:

$$MPR_i = \frac{\sum_{score_j<score_i}(f_j * r_j) + \frac{1}{2}\sum_{score_j=score_i}(f_j * r_j)}{\sum_i f_j}$$

Where *f* is the number of users who have seen the item, and *r* is the *percentile-ranking* of the item within the list. $r_i=0\%$ means that *i* is on top of the list, while 100% means that it is at the bottom.

2.1.6 Stochastic Gradient Descent

Finally, everything is ready for the actual work to begin. Just for a quick recap. At this point, *M* is normalized, *U* and *V* are initialized, and the order of the iteration is decided. The only missing piece is the actual change that should happen when each element of the matrix is visited. Again, the goal is to improve the prediction matrix with the change along a certain measurement.

The values in *U* and *V* should be changed in a way that the inspected prediction value comes closer to the known value. Notice, that for *P[i,j]* this means changing only the values in row *i* of *U*, and column *j* of *V*, as *P[i,j]* equals the product of these two. Improvement will happen using a technique called *Stochastic Gradient Descent*, or *SGD*. *SGD* can tell how should the values in U and V change to reduce error. In general, it's formula for SGD is:

$$w_{t+1} = w_t - l * \nabla_w Q(w_t)$$

Where *l* is the learning rate, and *Q(w)* is the "loss function". In this case, this translates to a less complex formula. The measurement the algorithm is meant to minimalize is *RMSE*, and thus the loss function is the error of each prediction:

$$err = \left(M_{ij} - P_{ij}\right)^2$$

Remember, that

$$P_{ij} = \sum_k U_{ik} V_{kj}$$

Next, the gradients should be determined for the related row of *U* and the related column of *V*:

$$\frac{\partial err}{\partial U_{ik}} = -2\left(M_{ij} - P_{ij}\right)V_{jk}^T$$

$$\frac{\partial err}{\partial V_{kj}} = -2\left(M_{ij} - P_{ij}\right)U_{ki}^T$$

This is where the name **Gradient Descent** comes from. *RMSE* is minimalized, using a gradient-based method: it is derived along the parameters, which results in the gradients. It is **Stochastic**, because random samples are taken from the training dataset. The result of all this is a formula for changing each of the relevant values of *U* and *V* during each round of improvement:

$$U_{ik} = U_{ik} + 2l\big(M_{ij} - P_{ij}\big)V_{jk}^{T}$$

$$V_{kj} = V_{kj} + 2l\big(M_{ij} - P_{ij}\big)U_{ki}^{T}$$

Note, that while most of these values are given, *l* can be of any value. Picking a good learning rate is of key importance in the algorithm. A badly chosen learning rate can make the prediction even worse.

2.2 Matrix Factorization with Alternating Least Squares

Matrix factorization with Alternating Least Squares (or *ALS*) isn't very different from the previous model. The process itself is the same, and most of the steps are also identical. An initial matrix is filled with the known values, the matrix is normalized, *U* and *V* are initialized, the values in them are improved, and then the normalization is restored. The only difference is the algorithm used for improving *U* and *V*, which is *ALS*.

2.2.1 Alternating Least Squares

The basic idea behind *Alternating Least Squares* is the following. Instead of making improvements one by one based on the known values, entire rows or columns are improved at the same time. Only one factor matrix is improved at a time – when *U* is improved, *V* is locked, and vice versa – this is why it is called **Alternating**. After choosing a row or column, the **Least Squares** problem must be solved.

There are two cases: either *U* or *V* is locked. In this more detailed description, a row (U_1) is improved, meaning *V* is locked. Once again, the measurement to be improved is *RMSE*. With known *n* items in the first row of the matrix, it is:

$$RMSE = \sqrt{\frac{(M_{11} - U_1V_1)^2 + \cdots + (M_{1n} - U_1V_n)^2}{n}}$$

RMSE is to be minimalized, meaning the solution is where the derivate of the above function is 0:

$$-2(M_{11} - U_1V_1)V_1 - \cdots - 2(M_{1n} - U_1V_n)V_n = 0$$

$$(M_{11} - U_1V_1)V_1 + \cdots + (M_{1n} - U_1V_n)V_n = 0$$

After some rearrangements:

$$M_{11}V_1 - (U_1V_1)V_1 + \cdots + M_{1n}V_n - (U_1V_n)V_n = 0$$

$$(U_1V_1)V_1 + \cdots + (U_1V_n)V_n = M_{11}V_1 + \cdots + M_{1n}V_n$$

Take *U* and *V* into elements, and the following system of equations is the result:

$$\begin{gathered} V_{11}(U_{11}V_{11} + \cdots + U_{1k}V_{1k}) + \cdots + V_{n1}(U_{11}V_{n1} + \cdots + U_{1k}V_{nk}) = M_{11}V_{11} + \cdots + M_{1n}V_{n1} \\ \vdots \\ V_{1k}(U_{11}V_{11} + \cdots + U_{1k}V_{1k}) + \cdots + V_{nk}(U_{11}V_{n1} + \cdots + U_{1k}V_{nk}) = M_{11}V_{1k} + \cdots + M_{1n}V_{nk} \end{gathered}$$

Where *k* is the common dimension of *U* and *V*.

After rearranging it:

$$\begin{gathered} U_{11}\left({V_{11}}^2 + \cdots + {V_{n1}}^2\right) + \cdots + U_{1k}(V_{11}V_{1k} + \cdots + V_{n1}V_{nk}) = M_{11}V_{11} + \cdots + M_{1n}V_{n1} \\ \vdots \\ U_{11}(V_{11}V_{1k} + \cdots + V_{n1}V_{nk}) + \cdots + U_{1k}\left({V_{1k}}^2 + \cdots + {V_{nk}}^2\right) = M_{11}V_{1k} + \cdots + M_{1n}V_{nk} \end{gathered}$$

Putting it into matrices:

$$\begin{bmatrix} \left({V_{11}}^2 + \cdots + {V_{n1}}^2\right) & (V_{11}V_{12} + \cdots + V_{n1}V_{n2}) & \cdots & (V_{11}V_{1k} + \cdots + V_{n1}V_{nk}) \\ (V_{11}V_{12} + \cdots + V_{n1}V_{n2}) & \left({V_{12}}^2 + \cdots + {V_{n2}}^2\right) & & (V_{12}V_{1k} + \cdots + V_{n2}V_{nk}) \\ \vdots & & \ddots & \vdots \\ (V_{11}V_{1k} + \cdots + V_{n1}V_{nk}) & (V_{12}V_{1k} + \cdots + V_{n2}V_{nk}) & \cdots & \left({V_{1k}}^2 + \cdots + {V_{nk}}^2\right) \end{bmatrix}$$

$$\begin{bmatrix} M_{11}V_{11} + \cdots + M_{1n}V_{n1} \\ M_{11}V_{12} + \cdots + M_{1n}V_{n2} \\ \vdots \\ M_{11}V_{1k} + \cdots + M_{1n}V_{nk} \end{bmatrix}$$

Or, in a more compact format:

$$\begin{bmatrix} \sum_{l=1}^{n} V_{l1}^2 & \sum_{l=1}^{n} V_{l1}V_{l2} & \cdots & \sum_{l=1}^{n} V_{l1}V_{lk} \\ \sum_{l=1}^{n} V_{l1}V_{l2} & \sum_{l=1}^{n} V_{l2}^2 & & \sum_{l=1}^{n} V_{l2}V_{lk} \\ \vdots & & \ddots & \vdots \\ \sum_{l=1}^{n} V_{l1}V_{lk} & \sum_{l=1}^{n} V_{l2}V_{lk} & \cdots & \sum_{l=1}^{n} V_{lk}^2 \end{bmatrix} \begin{bmatrix} \sum_{l=1}^{n} M_{1l}V_{l1} \\ \sum_{l=1}^{n} M_{1l}V_{l2} \\ \vdots \\ \sum_{l=1}^{n} M_{1l}V_{lk} \end{bmatrix}$$

This can be used as an input for the Least Squares problem. For the columns, the result is very similar, and can be reached the same way:

$$\begin{bmatrix} \sum_{l=1}^{n} U_{1l}^2 & \sum_{l=1}^{n} U_{1l}U_{2l} & \cdots & \sum_{l=1}^{n} U_{1l}U_{kl} \\ \sum_{l=1}^{n} U_{1l}U_{2l} & \sum_{l=1}^{n} U_{2l}^2 & & \sum_{l=1}^{n} U_{2l}U_{kl} \\ \vdots & & \ddots & \vdots \\ \sum_{l=1}^{n} U_{1l}U_{kl} & \sum_{l=1}^{n} U_{2l}U_{kl} & \cdots & \sum_{l=1}^{n} U_{kl}^2 \end{bmatrix} \begin{bmatrix} \sum_{l=1}^{n} M_{l1}U_{1l} \\ \sum_{l=1}^{n} M_{l1}U_{2l} \\ \vdots \\ \sum_{l=1}^{n} M_{l1}U_{kl} \end{bmatrix}$$

The problem must be solved for all rows and columns for *U* and *V*, possibly multiple times.

2.3 k-Nearest-Neighbors

k-Nearest-Neighbors or *kNN* is very different from the previous models in many ways. Its basis is the similarity between different items in the database. For each user-item pair, the prediction is based on those *k* items, that the user rated, and are most similar to the item in question.

2.3.1 Item distances

Item similarity is determined by their *distances*. This is where the name "nearest neighbor" comes from. The similar items are the ones that have the shortest distance from the item in question. Item distances can be defined in any way, but the way they are defined will have a great impact on how the model performs. When calculating distances, the 2 main questions are:

1. What should be the distance function?
2. What should be the basis for the distances?

They both greatly affect the outcome. Picking Euclidean distance or cosine distance yields different results. Similarly, picking the movies' genres as a basis, or the ratings the movie received by the users will have very different outcomes. Not only can these affect the accuracy of the predictions, but they can also greatly affect the performance of the model. Just consider this: if the basis for the distance is the genres of the movies, then an already calculated distance between two items isn't likely to change. However, if it is the ratings of the users, then it will change a lot. Since the most resource-consuming part of this model is calculating item distances, this should be taken into consideration.

For this model, *Pearson correlation* is used as a distance function, with the movies' genres as a basis. Pearson correlation can be calculated the following way:

$$\rho_{X,Y} = \frac{\mathrm{cov}(X,Y)}{\sigma_X \sigma_Y}$$

Where *X* and *Y* are the vectors that belong to the items, *cov* is the covariance, and σ is the standard deviation. As a first step in *kNN*, a distance matrix must be constructed, meaning the distance (in this case, the Pearson correlation) must be calculated and stored for every single item pair.

2.3.2 Prediction

When the distance matrix is ready, the model can start giving predictions. For any user-item pair, the algorithm for this is the following:

1. Out of the items *user* rated, get the *k* number of items, that have the smallest distance from *item* (based on the distance matrix)
2. Apply the following formula:

$$pred_{i,u} = \frac{\sum_{j \in mostSimilar} d(i,j) * r_u(j)}{\sum_{j \in mostSimilar} |r_u(j)|}$$

Where *d(i,j)* is the distance between *i* and *j*, and $r_u(j)$ is the rating user *u* gave to item *j*.

By using the movies' genres as a basis in this model, retraining times are greatly reduced. This way, not only it is not necessary to do a lot of recalculation on the item-distance matrix, but most of the recommendations can also be kept after an update. This is important, because unlike *matrix factorization* models, where producing the prediction matrix is very fast, producing all the predictions for *kNN* is extremely slow (1,000-10,000 times slower). This is not surprising. While the matrix factorization models are O(*users*items*factors*), *kNN* is proportional to the cost of calculating the similarity, and the number of ratings a user has. Unfortunately, making a global top list of items requires having all the predictions. This makes *kNN* a generally inferior choice in session-based cases. Using the genres as a basis also means that this model is only usable, if such data is available, otherwise ratings must be used as a basis. But for now, this is good enough for demonstrating the performance differences between the models.

3 Item-to-Item models

The previously introduced models recommend items to users based on their history. However, such history is not always available. Fortunately, there are other ways to give recommendations. Item similarities were already introduced with *kNN*. Using that, it is possible to give item recommendations, that are not connected to the user, but rather to the current item the user is looking at.

This section introduces 2 models. The first one is simple and fast, while the other one is more complex, and a lot slower.

3.1 Item-to-Item with Jaccard similarity

A fast way to calculate item similarities, is using *Jaccard similarity*. To do this, the only necessary information about the item, is the list of users, who are considered interested in it. Normally, *Jaccard similarity* is calculated the following way:

$$J(i,j) = \frac{|i \cap j|}{|i \cup j|} = \frac{|i \cap j|}{|i| + |j| - |i \cap j|}$$

However, to make sure no division by zero occurs, a modified version of this formula is used. This change does not modify the overall ranking of the items:

$$J(i,j) = \frac{|i \cap j|}{|i| + |j| - |i \cap j| + 1}$$

The elements of the item-to-item matrix must still be calculated, making this algorithm $\boldsymbol{O(I^2)}$, where $\boldsymbol{I}$ is the number of items known by the model.

3.2 Euclidean Item Recommender

A more complex approach is recommended by Koenigstein, N., & Koren, Y. [7]. The basis of this algorithm is the probability of a user looking at item *j*, after looking at item *i*:

$$P(j|i) = \frac{e^{-\|x_j - x_i\|}}{\sum_{l \neq i} e^{-\|x_l - x_i\|}}$$

Where x_k is item k's vector representation. If the item is not known by the model yet, then a vector is randomly generated. The aim of the algorithm is to maximize the *log-likelihood* of the training set.

$$\ln(P(j|i)) = \ln\left(\frac{e^{-\|x_j - x_i\|}}{\sum_{l \neq i} e^{-\|x_l - x_i\|}}\right) = \ln\left(e^{-\|x_j - x_i\|}\right) - \ln\left(\sum_{l \neq i} e^{-\|x_l - x_i\|}\right)$$

$$= -\|x_j - x_i\| - \ln(\sum_{l \neq i} e^{-\|x_l - x_i\|})$$

The vectors of i and j are changed whenever j is visited after i. Each element is recalculated using *Stochastic Gradient Ascent*. To do this, the derivative of $log(P(j|i))$ must be calculated for each dimension of the item vectors. To make the resulting formula less complex, taking the root when calculating the Euclidean distance is disregarded. It does not make any changes to the result, that couldn't be fixed by adjusting the *learning rate*.

$$\frac{\partial \ln(P(j|i))}{\partial x_{ik}} = -\frac{\partial \|x_j - x_i\|}{\partial x_{ik}} - \frac{\partial \ln\left(\sum_{l \neq i} e^{-\|x_l - x_i\|}\right)}{\partial x_{ik}}$$

$$-\frac{\partial \sum_{n=0}^{d} (x_{jn} - x_{in})^2}{\partial x_{ik}} - \frac{\partial \ln\left(\sum_{l \neq i} e^{-\|x_l - x_i\|}\right)}{\partial x_{ik}}$$

Where d is the length of the items' vector representation. Notice, that every dimension in the sum will result in 0, except for the k-th dimension:

$$2\left(x_{jk} - x_{ik}\right) - \frac{\partial \ln\left(\sum_{l \neq i} e^{-\|x_l - x_i\|}\right)}{\partial x_{ik}}$$

The derivative of the second part is more complex:

$$2\left(x_{jk} - x_{ik}\right) - \frac{\sum_{l \neq i} (e^{-\|x_l - x_i\|} \frac{\partial (-\sum_{n=0}^{d} (x_{ln} - x_{in})^2)}{\partial x_{ik}})}{\sum_{l \neq i} e^{-\|x_l - x_i\|}}$$

The case of sum inside the sum is the same as what was seem in the first part: it will be 0 for every dimension, except for k. This gives the result:

$$2\left(\left(x_{jk} - x_{ik}\right) - \frac{\sum_{l \neq i} (e^{-\|x_l - x_i\|} (x_{lk} - x_{ik}))}{\sum_{l \neq i} e^{-\|x_l - x_i\|}}\right)$$

Thus, the changes to the i item's vector will be the following:

$$x_{ik} = x_{ik} + 2l\left(\left(x_{jk} - x_{ik}\right) - \frac{\sum_{l\neq i}\left(e^{-\|x_l - x_i\|}(x_{lk} - x_{ik})\right)}{\sum_{l\neq i} e^{-\|x_l - x_i\|}}\right)$$

This means, that the algorithm must iterate through every item each time it updates a dimension of a vector.

While this makes updating the vectors slow, calculating the item-to-item matrix could be even slower. As usual, every element of the matrix must be calculated, already making the algorithm's complexity $O(I^2)$. However, since calculating P would also requires a full iteration through every item, the complexity is $O(I^3)$. Fortunately, since the denominator (the part that contains the iteration through all the items) is the same for every i, and the only usage of this matrix will be sorting the rows, calculating it is not necessary.

Determining the new vector if item j works in a similar way.

$$\frac{\partial \ln(P(j|i))}{\partial x_{jk}} = -\frac{\partial \left\|x_j - x_i\right\|}{\partial x_{jk}} - \frac{\partial \ln\left(\sum_{l\neq i} e^{-\|x_l - x_i\|}\right)}{\partial x_{jk}}$$

The result of the first part is the same, except for the sign, and for a while, the second part also looks similar:

$$-2\left(x_{jk} - x_{ik}\right) - \frac{\sum_{l\neq i}(e^{-\|x_l - x_i\|}\frac{\partial(-\sum_{n=0}^{d}(x_{ln} - x_{in})^2)}{\partial x_{jk}})}{\sum_{l\neq i} e^{-\|x_l - x_i\|}}$$

The result of this expression however is less complex. Notice, that in the case of i, the derivate was nonzero for every $n=k$. This time however, it will only be nonzero, if $n=k$, and $l=j$. This results in:

$$-2\left(x_{jk} - x_{ik}\right)\left(1 - \frac{e^{-\|x_j - x_i\|}}{\sum_{l\neq i} e^{-\|x_l - x_i\|}}\right)$$

Notice, that the last expression is $P(j|i)$. Thus, the result is:

$$-2\left(x_{jk} - x_{ik}\right)\left(1 - P(j|i)\right)$$

This leads to the new vector of item j:

$$x_{jk} = x_{jk} - 2l\left(x_{jk} - x_{ik}\right)\left(1 - P(j|i)\right)$$

It should also be noted, that this is still a simplified implementation, that only changes the vectors of *i* and *j*. Since the derivate of the other item vectors are also nonzero, they would also have to be changed. This would add a large amount of complexity to the algorithm, definitely making the update times unacceptable for the current use case.

4 Updating the models

After training the models for the first time, they should yield good results for a while, but this is bound to change over time. Not only user preferences can change, but new items, or new users will also appear. Because of this, the models must be updated over time. The strategy for this however, is not obvious. Many different aspects can be taken into consideration when working out a strategy.

It isn't an unrealistic expectation, to be able to assign different strategies to each model either. Or, to be able to change the update strategy on the fly. The role of the Update Proposer module is exactly this: any model can have an update proposer belonging to it. The update proposer monitors the model's activity, and when it should be updated based on its rule-set, it proposes the update.

4.1 Update Proposer Rules

Updating the models may be necessary for various reasons. The Update Proposer module makes it possible to create a set of rules for when this can happen. Generally, these rules revolve around one of four categories:

- Number of incoming messages
- Measurements getting worse
- Rate of unsuccessful predictions getting worse
- Number of times new users/items showed up

The following sections go more into detail about each of these. Note, that none of these rules are obligatory, the proposer can work with any subset of them.

4.1.1 Frequency-based rules

The first set of rules is the frequency-based ones. The only thing they take into consideration is the number of incoming messages since the last update. There are 2 rules that fall into this category: minimum number of messages necessary until the next update, and maximum number of messages allowed until the next update. If a minimum message limit is set, then it doesn't matter, if update should happen based on other rules, if the message count does not reach the minimum limit.

4.1.2 Measurement-based rules

The two measurements the updater takes into consideration are DCG, and recall. It does it in a way, that for each message, it stores these values, and calculates a moving average of them. If the rank is known, then the *DCG* is:

$$DCG = \frac{1}{log_2(rank)}$$

The number is elements (n) the predictor returns is known. In this case, recall is:

$$recall = \begin{cases} 0, if\ rank \geq n \\ 1, if\ rank < n \end{cases}$$

There 4 rules that fall into this category:

- Number of messages the moving average takes into consideration for DCG
- Number of messages the moving average takes into consideration for recall
- The minimum of what the moving average of DCG can reach
- The minimum of what the moving average of recall can reach

The first two does not actually propose any updates, they just affect how the last two work. In case the moving averages of DCG or recall fall under their respective minimum limits (and there were enough messages since the last update to fulfill the minimum messages necessary rule), an update is proposed.

4.1.3 Success-based rule

There is only one rule that falls into this category. The model can't always give predictions for a user-item pair, or a list of recommended items for a user. This is because the model may not have any information about the item or user in question. It can give back a fixed response, like 3 as a predicted rating, or the list of overall most popular items, but no real answer can be given that is exclusive to the user/item in question.

If this rule is used, then the proposer will keep track of the ratio of successful and unsuccessful predictions since the last update. If the rate of unsuccessful predictions reaches the limit (and there were enough messages since the last update), then an update is proposed.

4.1.4 Rules about new items

These rules don't propose updates either, they merely alter the dataset that is sent for updating. There are two of these:

- Minimum number of ratings necessary for an item to be forwarded for updating
- Minimum number of ratings necessary for a user to be forwarded for updating

If an update is proposed, but new item *i*, or new user *u* doesn't have enough ratings, then all the data belonging to them will be removed from the new dataset sent to the model. However, the update proposer will keep these, so if enough ratings will accumulate by the time the next update is proposed, they will be forwarded to the model.

5 Architecture

The following section contains a description of the architecture of the recommendation system, and the description of the whole system the recommendation system is in.

5.1 Recommender

The recommendation system consists of 3 main types of components:

- Models
- Update Proposer
- Framework

The first two were already covered. The following section contains a description of the Recommender Framework.

5.1.1 Framework

Many of the processes in the recommendation system are not exclusive to the models. While the way they are done can be very different, from an outsider's point of view, they are irrelevant. In some cases, it might also be necessary for the models to work together – for example, in case of having a combined model that uses results from all the other models. Because of this, it is better to have the models inside a framework, instead of storing them separately. It also forces the models to provide a common interface.

5.1.1.1 Model operations

There are several operations that the framework provides. These mostly forward the call to one or more models in a way that the framework is not actually aware of how each model works. This means any number of new models can be added to the framework if they provide the standard interface. Here follows a list of operations each model should be able to do.

Initialize models with data

By providing an initial ("training") dataset, the model must be able to initialize itself. Initialization includes everything, up to performing the first training of the model.

This step doesn't necessarily have to be fast. Since this must only be done once, and the system won't go live until this is done, this operation taking a long time is acceptable.

Receive data

In case a new rating comes in, the model must be able to process it. Along with the data, the model must also be able to receive and process the evaluation of the message. That is done by the framework, and includes the recall and DCG for the new item, and whether item and user were found. This information is forwarded to the update proposer of the model, which decides if the model should be updated, or not. If the answer is yes, then the model must start the update process.

Give prediction

When presented with a user-item pair, the model must be able to provide the following information:

- Prediction for the user-item pair
- The user was found in the model, or not
- The item was found in the model, or not

User ranking

When presented with a user id, the model must be able to provide the following information:

- A list of item ids, ranked by their predicted ratings for the user (in descending order)
- The user was found in the model, or not

Persist

The models must be able to provide all the data that is necessary to fully restore the model if necessary.

Restore

When provided with the data that *Persist* returned, the model must be able to fully restore itself to its former state.

5.1.1.2 Other functions

The framework doesn't only store and communicate with the models, it also communicates with the outside world. Upon initialization, its only parameter determines how long the recommendation list should be. It provides functions for both testing, and live operation. These include:

I/O operations on the disc

There are many reasons for the framework to write data to, and read data from the disc. Some of these are for testing/simulation: the sample can be acquired from the disc. Other uses however are also for live operation: the models are persisted to the disc. It is important to note that while persisting the models into a database was considered, it proved to be very ineffective.

Database operations

The system that the recommender is used in reads the recommendations from a database (mongoDB - https://www.mongodb.com). Because of this, the framework must be able to save recommendations into the database

Message queue operations

The framework can send messages to the message queue (RabbitMQ - https://www.rabbitmq.com), but that functionality is only used for testing. Receiving messages however is very important. This is how the recommender learns about new ratings.

5.2 System-wide architecture

This section introduces the architecture of system that the recommender is used by. Note, that this does not contain every component of the system, only the ones that are related to the recommender.

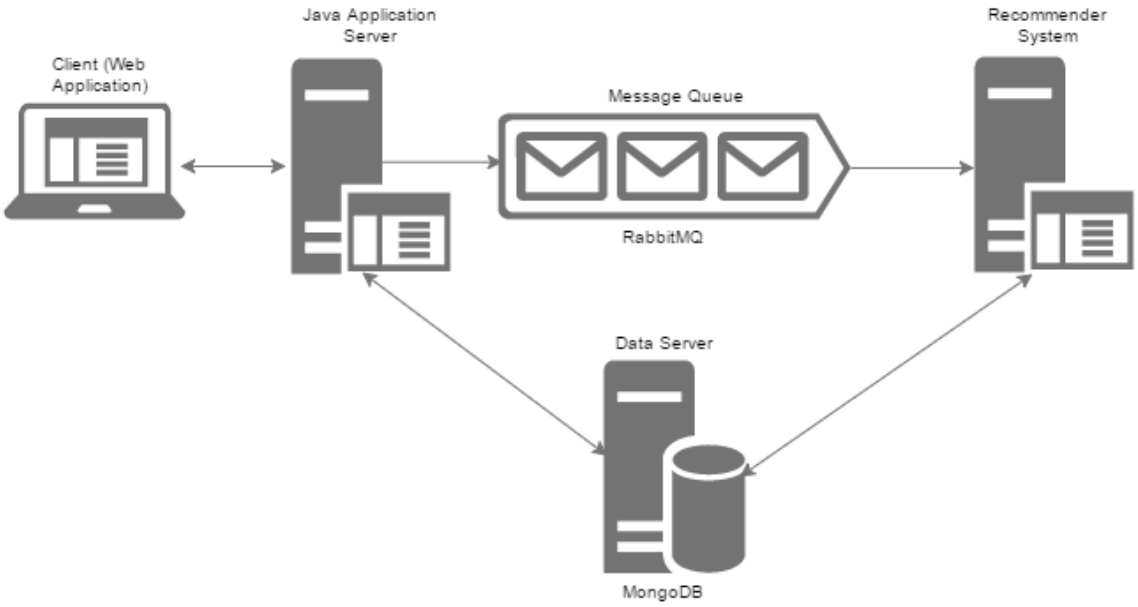

Fig. 1: System-wide architecture

The recommender has already been introduced. The next part contains a short introduction of the client and the java application. For information on the message and storage formats, see the Interfaces section of the Appendix.

5.2.1 Client

This is where the data for the recommender originates from, and this is where the recommendation will arrive to in the end. While currently the client only sends one type of message to the server that is used by the recommender, this can (and in the future, probably will) change. A recommender can get a wide variety of messages from the clients, that can be interpreted in many ways. This can range from the very explicit “rating”, “favorite”, or “open”, to “time spent looking at the item”, or even data derived from mouse movement.

5.2.2 Java application

The Java application provides a *REST* [10] interface for the client. *REST* (*REpresentational State Transfer)*, or *RESTful* web services are a way of making

communication between computer systems possible. It is stateless, uses a client-server model, and offers a uniform interface.

Although this is the biggest component of the whole system, the part that is related to the recommender is small. In case the client sends a request that should change the rating of an item, the Java application does two things:

- Save the newly acquired data into the database;
- Send the new data via message queue to the recommender.

The java application can also receive requests from the client for recommendations. In this case, it reads the entry from the database that belongs to the user that made the request. If the database contains no such information, it reads the recommendation belonging to the id ‘top’. For item recommendations, the id of the item identifies the recommendation list.

6 Implementation

After learning about the models, the update proposer, and the framework in theory, it is time to look into the implementation. My language of choice for this was *Pyhton 3*. I used several libraries to implement the algorithms. I used *pandas* to handle the input data, *pika*, to handle *RabbitMQ*, and *pymongo*, to handle *MongoDB*. But most importantly *numpy*, to make most of the necessary calculations.

6.1 Models

Like previously, the first part of the recommender to be introduced is the models. As mentioned before, any model that the framework should handle must have a predefined set of operations available. Each model has its own class.

There are a few things, that are common in all the models. The parameters of the underlying algorithm (which can be found in the appendix) can be set at instantiation. Since creating a matrix is necessary, maps (or as they are called in Python, *dictionaries*) are created, that can tell which row/column of the matrix belongs to which user/item. This is especially useful upon persisting and restoring the data for the models. Storing the matrix itself as a dictionary was considered, but the performance was very poor compared to using a numpy array.

6.1.1 Matrix Factorization with Stochastic Gradient Descent

The first step of Matrix Factorization is initializing the matrix of known rating values. For this purpose, a numpy array full of zeros is created with USERSxITEMS dimensions. After that, the known values are set one by one. The data is zipped for performance reasons, iterating through the pandas DataFrame (*"loaded"*) would be a lot slower. The variables *userRows* and *contentRows* contain which id belongs to which row/column in the matrix.

```
self.M = np.zeros((len(userIds), len(contentIds)))
for r in zip(loaded['ownerId'], loaded['contentId'], loaded['rating']):
    self.M[userRows[r[0]], contentRows[r[1]]] = r[2]
```

As mentioned before, the next step is preprocessing. This also has multiple steps, the first of which is normalization. The algorithm that normalizes the rows work exactly as it has been described earlier. Row by row, it takes the average of the nonzero elements,

then subtracts this average from all these elements. In the end, the average of each row is returned, since that information is necessary for restoring the matrix after the training is complete.

```
u = np.zeros(M.shape[0])
for i in range(M.shape[0]):
    m = M[i]
    elems = m[m != 0]
    if len(elems) == 0:
        u[i] = 0
    else:
        u[i] = np.mean(elems)
    mask = M[i] != 0
    M[i, mask] = np.subtract(M[i, mask], u[i])
return u
```

Normalizing the columns doesn't require a separate function with similar logic, since it can be done by passing a transposed matrix to the function normalizing the rows. The next step in preprocessing is filling up U and V. For this, using numpy's *random.rand* function is a good choice.

```
div = a/d
base = float(math.sqrt(abs(a/d)))
base = math.copysign(base, div)
if c == 0:
    M.fill(base)
else:
    M = (np.random.rand(M.shape[0],M.shape[1]) - 0.5) * 2*c + base
return M
```

Finally, a function that returns a random permutation of known elements is also necessary. Fortunately, numpy can tell the coordinates of all the nonzero elements. By shuffling this list, a random permutation can be created.

```
nonzeros = np.transpose(np.nonzero(M))
np.random.shuffle(nonzeros)
return nonzeros
```

After preprocessing is done, improving the matrix can begin. As discussed before, the complexity here is relatively low. With an *M* matrix, improving *U* and *V* based on the coordinates *(i;j)*, with a learning rate of *l* can be done in just a few lines:

```
p = np.copy(U[i])
V = V.transpose()
q = np.copy(V[j])
err = M[i,j] - np.dot(p, q)
U[i] = p + 2*l*err*q
V[j] = q + 2*l*err*p
V = V.transpose()
```

The only thing that remains is performing the training. This is done by executing the improve function *runPerRound* times on each coordinate, specified by the permutation. As a last step, it is important not to forget to restore the normalization on the resulting prediction matrix.

```
for n in range(self.runPerRound):
    for p in perm:
        self.improve(M, U, V, p[0], p[1], self.l)

P = np.dot(U,V)
P = self.restoreNormalization(P, rowAvg, colAvg)
```

Restoring the normalization is mostly taken care of by *numpy*:

```
P = np.add(P, columnAvg)
P = P.transpose()
P = np.add(P, rowAvg)
P = P.transpose()
return P
```

Updating the model follows the same exact logic, as the initial training, except it does not generate new U and V matrices, but uses the old ones. If new users or items have been added to the model, new random values are generated for those, and those only. Those new elements are then appended to the matrices.

By having the prediction matrix, the model can already give predictions, and recommendations. When giving predictions, the model must look up which value belongs to the requested user-item pair in the prediction matrix. If the user, or the item is not known by the model, then 3 is returned. if the prediction is less than one, or more than 5, then 1 and 5 are returned respectively. Booleans are also returned that indicate if the user and the item were found, or not. This information is not necessary for the algorithm itself, but it can be used by the update proposer, and is also logged.

```
if (userId not in self.uRows) or (contentId not in self.cRows):
    return (userId in self.uRows, contentId in self.cRows, 3)
uRow = self.uRows[userId]
cRow = self.cRows[contentId]
pred = self.P[uRow,cRow]
if pred > 5:
    pred = 5
if pred < 1:
    pred = 1
return (True, True, pred)
```

Giving recommendations is a bit less straightforward. The row of the user that requested the recommendation should be sorted, but since the matrix only contains

numbers, first the data from the row should be put in a structure that contains the item ids for each prediction. After that, the sorting can happen. Finally, the first *n* items should be returned. If recommendation is not available for the user, then a global recommendation should be returned. A Boolean is also returned, that tells if giving user-specific recommendation was successful, or not. This server as a similar purpose, as in the case of giving predictions. One last thing, that should be noted, is that the recommendations are cached in the model. While in the live system, where recommendations are taken from the database, this is not used, but during simulations where the database was not used, this could speed things up a lot.

```
userRanking = list()
if uId not in self.uRows:
    return (False,self.getTopRanking())
userId = self.uRows[uId]
i = 0
for v in self.P[userId]:
    userRanking.append((self.reverseContentRows[i], v))
    i += 1
userRanking.sort(key=lambda r: r[1], reverse=True)
self.userRanking[uId] = userRanking
return (True,userRanking[0:n])
```

The global top ranking is calculated similarly to the user specific ranking, the only difference is the input data. Instead of taking one of the rows of the prediction matrix, a vector is taken, that contains the averages of the columns of the prediction matrix.

```
colRank = list()
for c in self.P.transpose():
    colRank.append(np.mean(c))
```

The last thing that should be discussed about this model is persisting and restoring the data stored in it.

Persisting this model requires saving the following data:

Attribute	*Description*
M	The original matrix containing the ratings of the users
U	The factor matrix that belongs to the users
V	The factor matrix that belongs to the items
rowAvg	The averages of the ratings of the users
colAvg	The averages of the ratings of the items
userRows	A map of which user belongs to which row
contentRows	A map of which item belongs to which column

Restoring most of the data is self-explanatory. The only exception is the prediction matrix itself. The product of U and V does not yield the proper *P* matrix yet, because U and V were created for a normalized *M*. The prediction however should be given from a non-normalized matrix. Because of this, after producing U and V, the normalization must be restored. This is the reason the averages of the rows and columns had to be persisted.

```
self.P = np.dot(self.U,self.V)
self.P = self.restoreNormalization(self.P,self.rowAvg,self.colAvg)
```

6.1.2 Matrix Factorization with Alternating Least Squares

Most of the code from the previous model can be reused as is for Alternating Least Squares. This should not be surprising, as it was mentioned before, the only real difference is the way the factor matrices are improved. The improve methods are called in a similar way, as they were before:

```
for n in range(self.runPerRound):
    for i in range(M.shape[0]):
        U = self.improveRow(M,U,V,i)
    for j in range(M.shape[1]):
        V = self.improveCol(M,U,V,j)
```

The real difference is in the functions *improveRow* and *improveCol*. Since they are very similar, only the implementation of *improveRow* is included here:

```
p = np.copy(U[row])
Vt = V.transpose()
cols = np.nonzero(M[row])[0]
vs = Vt[cols]
a = [[sum([Vt[col,j]*Vt[col,i] for col in cols]) for i in
range(len(Vt[0]))] for j in range(len(U[0]))]
b = [sum([M[row,col]*Vt[col,i] for col in cols]) for i in
range(len(U[0]))]
U[row] = np.linalg.lstsq(a,b)[0]
return U
```

In this case, the *row*th row of the U matrix is being improved. The variable *cols* contains the columns, where the selected row of the matrix has a nonzero value. These columns of the V matrix will be considered. The variables *a* and *b* are the matrices that serve as input for the *Least Squares* problem. Calculating the solution of the problem is the responsibility of numpy. The selected row of the U matrix is finally replaced with the solution.

6.1.3 k-Nearest-Neighbors

The initial matrix creation works the same way for k-Nearest-Neighbors, as it works for matrix factorization: a matrix of zeros is created, and then it is filled in with the known values. However, *kNN* requires a second matrix, that stores the item distances. This is a matrix with ITEMSxITEMS dimensions, with every value calculated one by one. As mentioned before, this value is the Pearson correlation between the two items.

```
def standardDeviation(self,values):
    m = np.mean(values)
    s = np.sum(np.square(np.subtract(values,m)))
    return math.sqrt((1/(len(values)-1)*s))

def covariance(self,x,y):
    xm = np.mean(x)
    ym = np.mean(y)
    xx = np.subtract(x,xm)
    yy = np.subtract(y,ym)
    s = np.sum(np.multiply(xx,yy))
    return (1/(len(x)-1))*s

def pearson(self,x,y):
    sxy = self.covariance(x,y)
    sx = self.standardDeviation(x)
    sy = self.standardDeviation(y)
    return sxy/(sx*sy)
```

Most of the logic is the same as before. Predictions are calculated, and recommendations can be given based on these predictions. The only missing piece here

is calculating the prediction. The first step is taking the list of items that the user rated. This can be done by taking the coordinates of the nonzero elements of the user's row.

```
userRow = self.M[self.uRows[userId]]
nonzero = np.nonzero(userRow)[0]
```

Next, out of these items, the (at most) *k* items should be taken, that are most similar to the item, that the prediction is made for.

```
for v in rated:
    if v != cRow:
        rank.append((str(self.reverseContentRows[v]),
contentDistances[v]))
rank.sort(key=lambda r: r[1], reverse=True)
rank = rank[0:self.k]
return rank
```

Finally, the previously introduced formula should be applied to the returned list of items.

```
userRow = self.M[self.uRows[userId]]
num = sum([x[1]*userRow[self.cRows[x[0]]] for x in movies])
den = sum([abs(x[1]) for x in movies])
knn = num/den
return knn
```

As mentioned, filling up the entire prediction matrix is slow, since the values must be calculated one by one. Because of that, giving recommendations can also be very slow, since the entire row must be calculated for it. Giving global recommendations is extremely slow, because it requires the entire prediction matrix to be calculated. For these reasons, while this model can be good for giving predictions, it should only be used for giving recommendations, if long training times are not an issue.

Persisted attributes are the following:

Attribute	*Description*
M	The original matrix containing the ratings of the users
D	The matrix containing the item distances
userRows	A map of which user belongs to which row
contentRows	A map of which item belongs to which column
similarityBase	Any attribute that stores the information, that serves as basis for the items' similarities. In this case, it is the genres.

6.1.4 Item-to-Item with Jaccard similarity

The basis of this algorithm is keeping track of which users saw which items. For this purpose, a dictionary of item ids is created. The values in the dictionary are sets. When a user sees the item, then the user's id is added to the set. The id of the item that each user saw last is also kept track of, but this is solely for testing purposes.

```
for d in zip(loaded['ownerId'], loaded['contentId']):
    self.lastSeen[d[0]] = d[1]
    if d[1] not in self.items:
        self.items[d[1]] = set()
    self.items[d[1]].add(d[0])
```

Once again, an item similarity matrix is created. It has ITEMSxITEMS dimensions, and the stored values are the Jaccard similarities between the two items. Fortunately, calculating Jaccard with sets is very fast.

```
def jaccard(self,i,j):
    ij = i & j
    return len(ij)/(len(i)+len(j)-len(ij)+1)
```

The item-to-item recommenders don't give predictions, only recommendations. This works exactly as before, except this time the rows of the matrix don't represent users, but items.

The attributes that must be persisted are the following:

Attribute	*Description*
M	A matrix that contains the item similarities
contentRows	A map of which item belongs to which row/column
lastSeen	A map of the items that each user visited last (for testing)
items	A map of items, that store which users visited each item

6.1.5 Euclidean Item Recommender

The *EIR* algorithm keeps track of 3 things. The first one is the item that was last seen by each user. In this case, this is not only for testing: if a user visits item *j* after item *i*, then the items of these vectors must be updated. The second one is the item vectors themselves. The last one is the list of users who saw each item. In this case, the last one is not necessary for the algorithm, it is only needed for testing/evaluation purposes.

Every time a new item is added to the system, a random vector is generated for it, with the specified size. The values in the vector are in the range of [0, 1).

```
items[item] = np.random.rand(d)
```

The basis of this algorithm is calculating *P(j|i)*. Numpy has a built-in function for calculating the Euclidean distance (*linalg.norm*), so it doesn't have to be done by hand. The *exp(n)* function returns e^n. The formula translates to the following code:

```
def P(j,i):
    num = np.exp(-np.linalg.norm(items[j]-items[i]))
    den = sum([np.exp(-np.linalg.norm(items[l]-items[i])) for l in items
if (l != i)])
    return num/den
```

To improve the vectors, the gradients of the log-likelihood must be calculated. The same functions can be used to help with the calculation in both cases.

```
def calculateDiffI(i,j,k):
    diff = items[j][k] - items[i][k]
    num = sum([np.exp(-np.linalg.norm(items[l]-items[i]))*(items[l][k] -
items[i][k]) for l in items if (l != i)])
    den = sum([np.exp(-np.linalg.norm(items[l]-items[i])) for l in items
if (l != i)])
    result = 2*(diff-num/den)
    return result
```

Since the end of the gradient for vector *j* is the same as *P(j|i)*, I used the function that calculates *P*.

```
def calculateDiffJ(i,j,k):
    diff = items[j][k] - items[i][k]
    p = P(j,i)
    result = -2*diff*(1-p)
    return result
```

When changing the vectors, this change must be done for every element of the vector. As before, *l* is the learning rate.

```
for k in range(len(i)):
    diffI = calculateDiffI(i,j,k)
    diffJ = calculateDiffJ(i,j,k)
    items[i][k] = items[i][k] + l*diffI
    items[j][k] = items[j][k] + l*diffJ
```

With all this, the vectors are already ready to be improved. The real time-consuming part however, is still ahead. The item similarity matrix must be filled up once again, but in this case, each element of the matrix is the nominator of *P*. After the matrix is ready, giving recommendations for items work exactly as it works in all the other cases.

```
l = len(items)
M = np.fromfunction(lambda i,j: np.exp(-
np.linalg.norm(items[reverseCRows[j]]-items[reverseCRows[i]])), (l,l))
```

When persisting the model, the following attributes must be saved:

Attribute	*Description*
lastSeen	A map of the items that each user visited last
items	The vectors of the items known by the model
M	A matrix that contains the item similarities
itemsSeen	A map of items, that store which users visited each item (for testing)
contentRows	A map of which item belongs to which row/column

6.2 Update Proposer

The properties of the update proposer (described in 4.1) can be set in the constructor. Apart from these values, the proposer must also store extra information.

First, and most importantly, the proposer has a *queue*. This is where all the messages are stored, that have not been sent to the model yet. They are waiting for an

update to happen, but until then, it is the responsibility of the proposer, to keep track of them.

To work with rules about unknown elements, 3 attributes had to be added. The first one, *unknownCount* keeps track of how many unsuccessful predictions happened since the last update. The other two, *unknownItems* and *unknownUsers* are dictionaries. If an unsuccessful prediction or recommendation happens, then the unknown user or item is put in these dictionaries. The values that belong to the ids are the number of times the new item/user has been referenced. The messages that use these values are only forwarded, if that number is greater than, or equals the one set in the rules.

There are also numpy arrays for storing a predefined number of measurements, for calculating their moving averages for measurement-based rules.

When a new message arrives, the following things happen in the proposer:

1. The message is added to the queue, and the message counter is incremented.
2. If the prediction/recommendation was unsuccessful, the *unknownCount* is incremented, along with the proper values in *unknownItems* and/or *unknownUsers*.
3. If the prediction/recommendation was successful, and the item is considered "liked" by the user (in test cases, it means a rating of 4, or better), then the measurements are saved.
4. Based on the rules, it is decided, if an update should happen.
5. If it should, then the queue is replaced with the list of messages, that should not be forwarded yet due to low message count belonging to the new user/item. The dictionaries, that keep track of this info are replaced with ones, that only contain these entries. Finally, the counters are reset, and the prepared messages are forwarded to the model.

Perhaps point 3. should be described in more detail. The following code shows how recall is handled:

```
if (userKnown and itemKnown) and (data['rating'] >= 4):
    self.recalls[self.recallCounter%len(self.recalls)] = recall
    self.recallCounter += 1
    if self.recallCounter == len(self.recalls):
        self.recallCounter = 0

if self.recallCounter < 0:
    recalls = self.recalls[0:self.recallCounter%len(self.recalls)]
    if len(recalls) == 0:
        currentRecall = 0
    else:
        currentRecall = np.mean(recalls)
else:
    currentRecall = np.mean(self.recalls)
```

Since a moving average is calculated, mod is used to place the new values in the fix sized array. While the calculations are straightforward after the counter reaches the size of the array, the messages before that must be handled with extra care. As a solution, initially the counter starts from -recallArrayLength instead of 0. In this case, when calculating the average, only the beginning of the array should be considered.

6.3 Framework

Since, the framework was mostly introduced in 5.1.1 this chapter only contains a few short implementation-related details.

The only parameter in of the constructor is the number of items the recommender should return. The models are stored in a dictionary. The key in this dictionary will be treated as the model's name, and will be used as filename for logging. The framework also has a queue and a lock. This is needed, because messages need to be taken from the message queue immediately, but it is possible, that they can only be processed later. The messages are put in the framework's own queue. If processing was not in progress yet, then it is started, and it will keep running while there are messages in the queue.

```
def processMessages(self,ch, method, properties, body):
    data = json.loads(body.decode())
    self.q.put(data)
    if not self.lock.locked():
        t = threading.Thread(target=self.process)
        t.start()

def process(self):
    self.lock.acquire()
    while not self.q.empty():
        data = self.q.get()
        evaluation = self.evaluate(data)
        self.sendDataToModels(data, evaluation)
        sys.stdout.flush()
        self.q.task_done()
    self.lock.release()
```

The messages are decoded using the *json* library, are put in the queue, and if the lock is not on, then a new thread is created, that's purpose is processing the messages in the queue. The lock is acquired, and messages are evaluated and sent to the models. The evaluation process consists of calculating measurements, and logging all the data from the massage, that is important for analysis. The models themselves will do very little with the data and the evaluation, as the models will hand them to their update proposers almost immediately. After all the messages in the queue have been processed, the lock is released.

7 The simulations

To test how the recommendation engine performs, some pre-existing data is necessary. For this purpose, my main data of choice was the *MovieLens* dataset (https://movielens.org/). It comes in different sizes. The simulations use the one with ~1.000.000 ratings (which contain timestamps). It is a good compromise, the dataset is big enough for testing various attributes of the system, but small enough to work with even on home computers.

The appendix contains more detailed description on the dataset.

7.1 Collaborative models

Using the parameters found in the Appendix, a simulation was run. This had 2 major parts:

1. Training period with the first 10% of the data
2. Test period with the other 90% of the data

The times required for training each model can be very different. This is how long the initial training took for each model:

Model	*Training time (s)*
Matrix Factorization with SGD	31
Matrix Factorization with ALS	71
k-Nearest-Neighbors	1127

The training time necessary for *kNN* is visibly much higher, than the other two. This is because the item-distance matrix must be fully calculated for this model. This is also the reason *kNN* was set in a way that the basis for this matrix is the genre of the items. If the basis was a constantly changing data, like ratings, then this matrix would always have to be recalculated, meaning every update would take similarly long time.

The testing period took a longer time. Using the settings seen in the Appendix, it ran for ~13 hours.

The update statistics for the 3 models were very similar. The DCG and the recall criteria were usually fulfilled just after reaching the minimum number of required

messages. The “rate of unknown items” criterion was also usually fulfilled in the first half of the simulation, but by the second half, it stopped triggering updates.

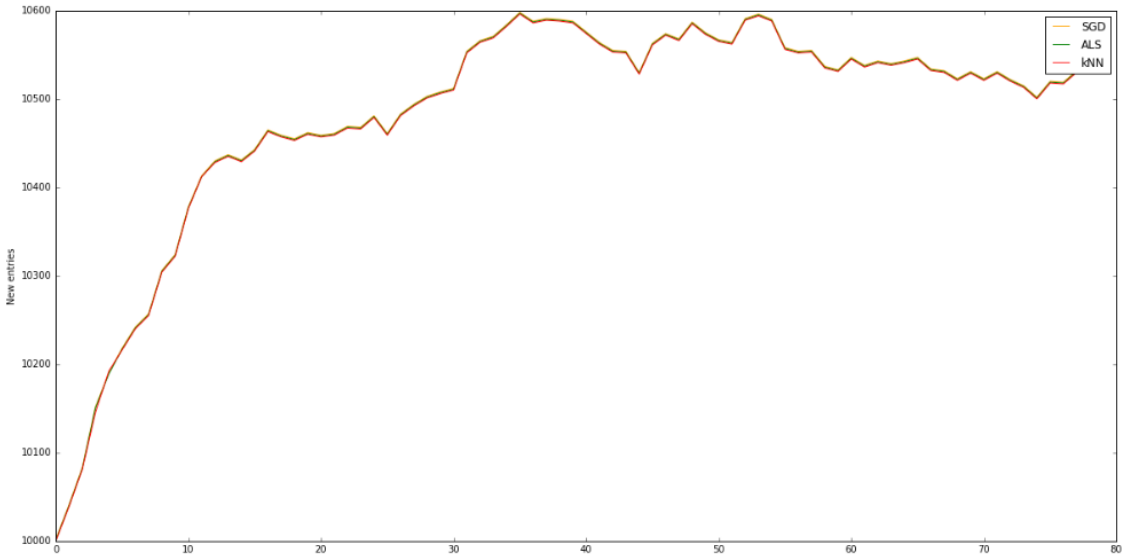

Fig. 2.: Number of new ratings put in the models after each update

As it is visible on Fig. 2, the number of new ratings that were put into the models with each update never reach very high numbers. If it wasn’t for elements that were kept in because not enough ratings belonged to a new user or item, this would probably be a constant 10.000.

It is also worth to look at the update times.

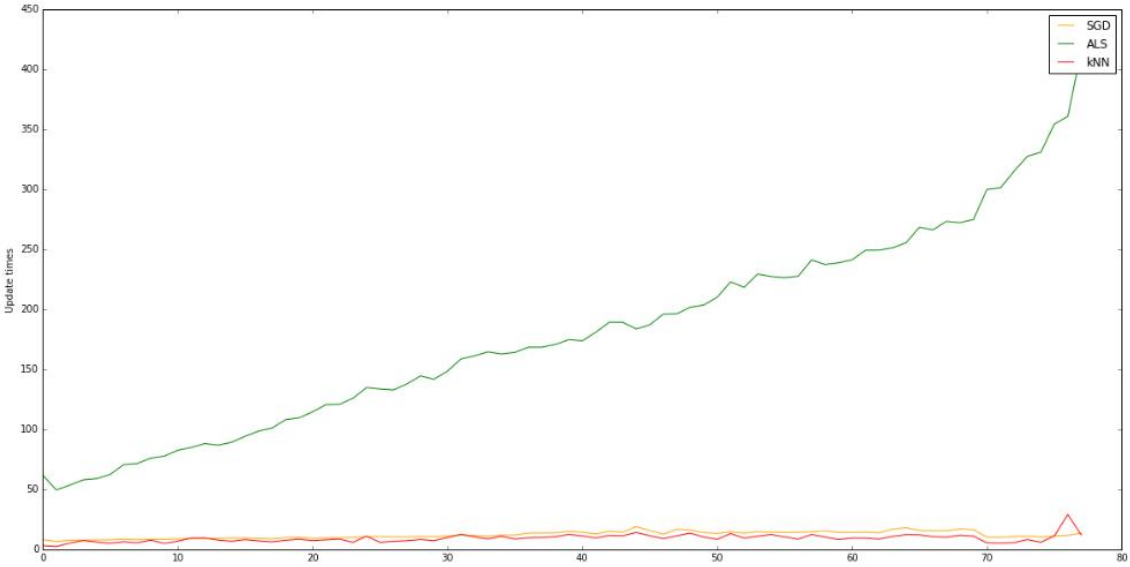

Fig. 3.: Update times (s) for each model.

On Fig. 3, it is visible that while *MF with SGD* (orange), and *kNN* (red) have a steady update time, *MF with ALS* requires significantly more time as the number of elements grow in the model. However, it should be noted, that this training time is still a lot lower, than what *kNN* would do, in case the item-distance matrix had to be recalculated.

All three models are evaluated along 3 measurements: RMSE, DCG, and recall. For RMSE, every rating is considered, but for DCG and recall, only the ones that are 4, or better. This is done to avoid getting worse measurements because of accurate recommendations. If an item, that was rated as 2 by the user would be considered, it would mean a low DCG and recall, even though the model successfully predicted that the user will not like the item.

The first measurement is RMSE. The moving average of 1.000 elements is shown on the chart.

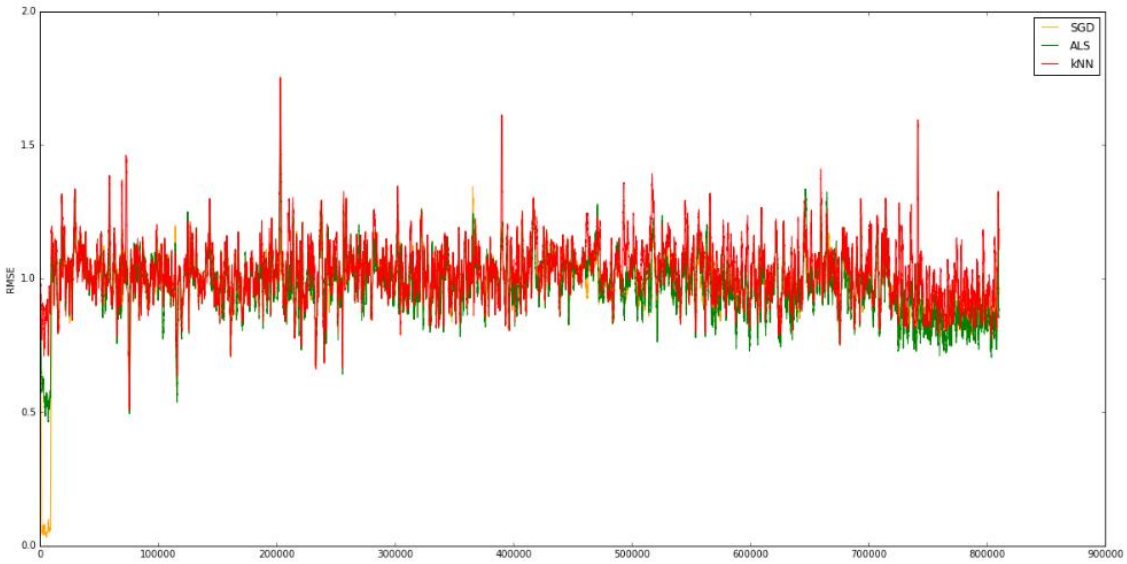

Fig. 4.: Moving average of RMSE for the 3 models

Out of the three models, *kNN* constantly has the highest RMSE, but isn't much higher than the other two. The other two seem to move around the same values, with *ALS* being maybe a bit better. However, with such a small difference, picking the model with *SGD* is probably a better idea, since it visibly scales better.

The next measurement is recall. The way recall should be evaluated is not trivial. For each single element, it is determined by the rank of the item. The question is: which elements should be considered? For an item that the user rated as 2 (if the models work well), it is expected that the item should be very low on the list. If that is the case, it shouldn't be in the "recommended" list. Such items should not worsen the average for recall. For this reason, only elements with a rating of 4, or higher are considered.

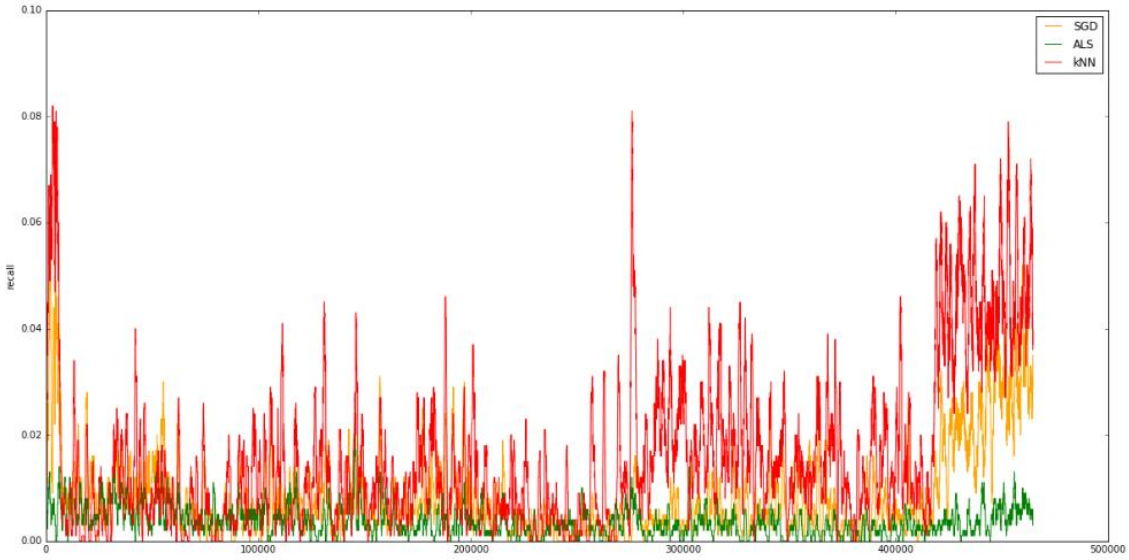

Fig. 5.: Moving average of recall for the 3 models

The results here are the complete opposite as what was seen for RMSE. *kNN* achieved the highest recall on average, with *ALS* getting the worst results. However, it should be remembered that *kNN* does not give global recommendations, since that would require calculating the full user-item matrix, which would raise the training time by magnitudes. If that would not be a concern, then *kNN* would probably be the best choice here, but since supporting sessions is one of the main goals of this recommender, that is not acceptable here. The clear choice of the other two is *MF with SGD*, which not only has a better recall, but also scales better.

The last measurement is DCG. As before, only elements with a rating of 4 or higher are considered.

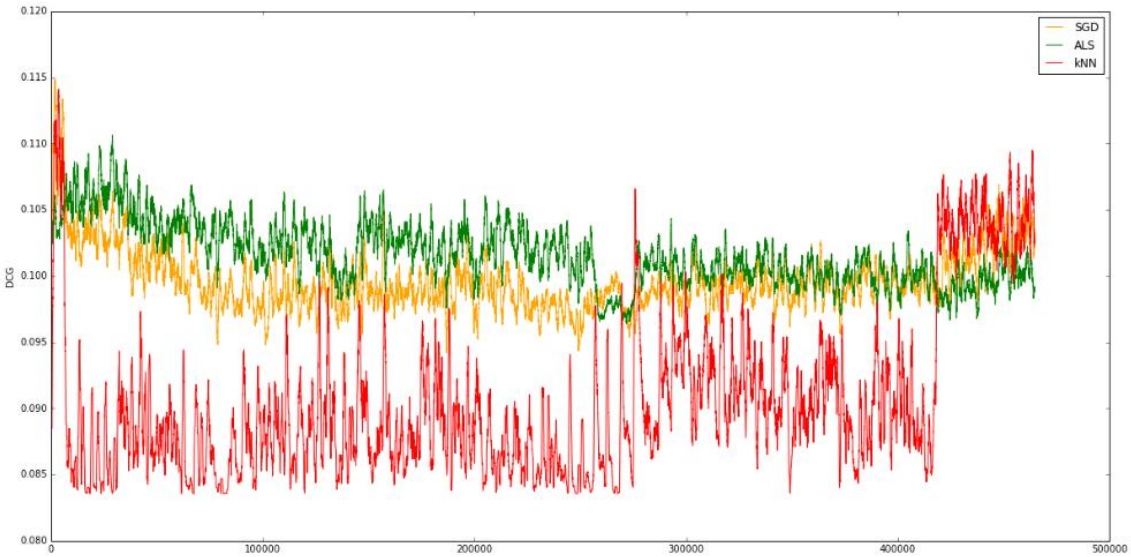

Fig. 6.: Moving average of DCG for the 3 models

In this case, *kNN* is all around the place, but most of the time it is deeply below the other two models. From the Matrix Factorization models, it appears that *ALS* usually gets better results. Perhaps these differences are big enough that if the focus is on optimizing for DCG, then scaling worse than *SGD* can be acceptable.

7.2 Item-to-Item models

Due to the high complexity of the *EIR* algorithm, a smaller data set was used for testing the item-to-item cases. The training dataset had 10.000 elements (for 2.096 items), while the test dataset had 90.000. Once again, the models' parameters can be found in the appendix. The difference in the complexity of the algorithms greatly show in the training times:

Model	*Training time (s)*
Item-to-Item with Jaccard	11
Euclidean Item Recommender	2702

In many cases, long training times (even with a relatively low item count) make this implementation of *EIR* an unacceptable choice. Some systems might have millions of items. Still, with a lower item count, with proper architecture, it can be a good choice, if it gives better results. Unfortunately, this does not seem to be the case.

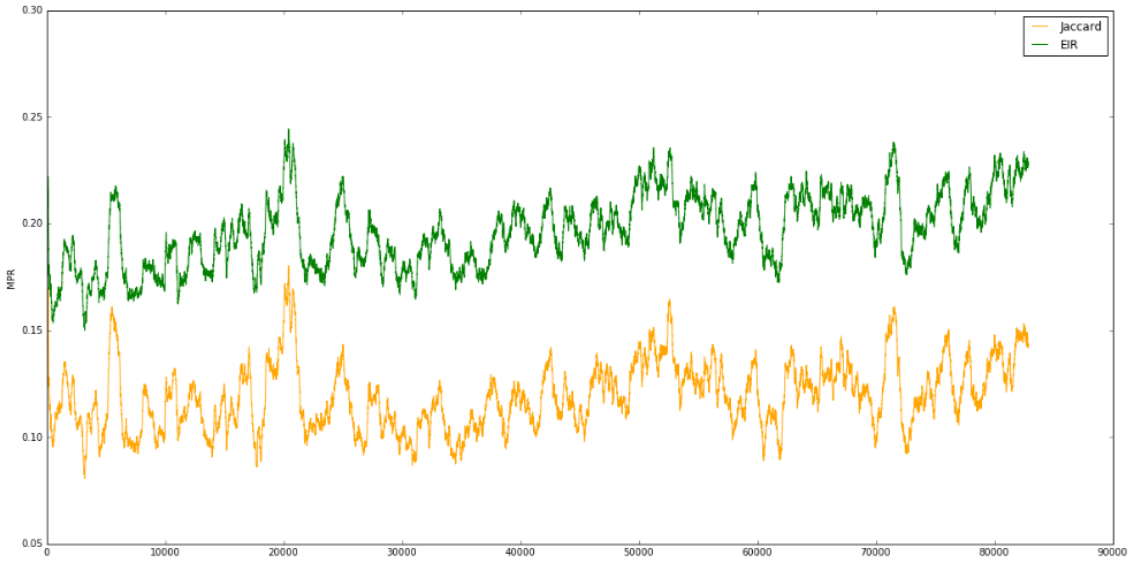

Fig. 7.: Moving average of MPR of the two models (MovieLens)

For evaluating *EIR*, the suggested metric in [7] was *MPR*, so I also use that to evaluate the performance of these models. In this case, the moving average (for 1.000 elements) of the MPR can be seen for both algorithms on Fig. 7. The results are also filtered, cases where no recommendation could be given are not included.

The MPR for *EIR* is visibly higher, than for the algorithm using Jaccard. I tried experimenting with different learning rates, but the MPR was always higher for *EIR*. One source of poor performance could be the size of the dataset. It is possible, that *EIR* requires more user input, until it starts working properly. But if that is the case, it would still mean that the set of cases, where *EIR* might be considered as the superior choice is even more limited.

It might be worth experimenting with other datasets though, that might reflect different user behavior. The next test is executed on a subset of the Netflix dataset, that was published for the Netflix Prize competition [8]. The size of the subset is the same as in the previous instance: 10.000 elements for training, and 90.000 for testing. However, the models (especially *EIR*) are expected to train for much longer, since this dataset contains 3.914 items.

Model	*Training time (s)*
Item-to-Item with Jaccard	35
Euclidean Item Recommender	4124

The results are a bit different in this case, but the main point stays the same.

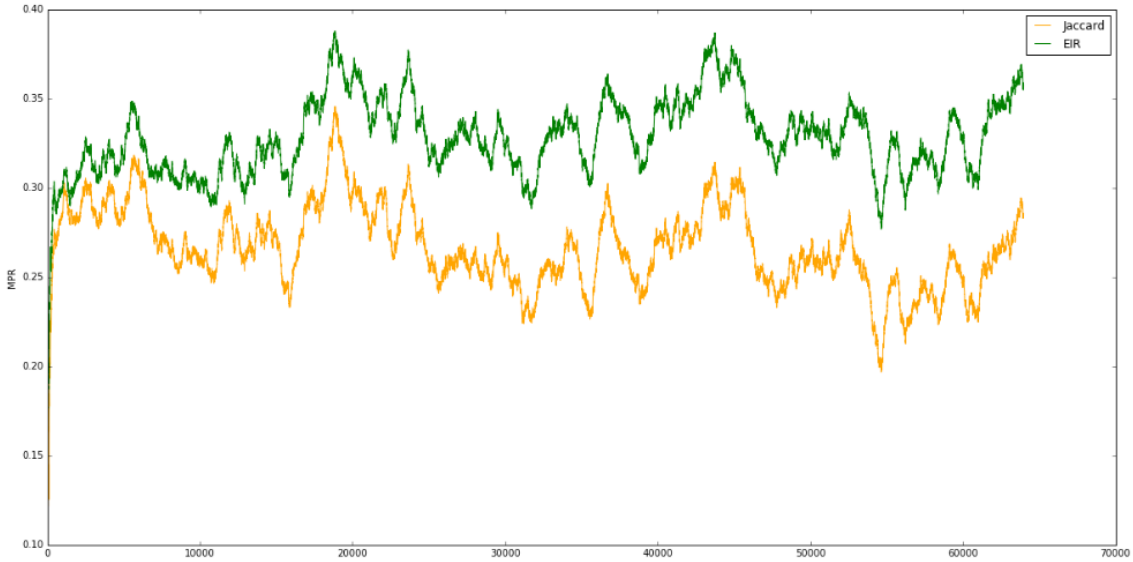

Fig. 8.: Moving average of MPR of the two models (Netflix)

The biggest difference is that in the beginning, the performance of the two models are almost equally good. As time passes however, the difference grows, and *EIR* starts having a significantly higher *MPR*. There are also less successful recommendations, but this ratio is still acceptable.

The last dataset contains books, and was collected by Cai-Nicolas Ziegler (http://www2.informatik.uni-freiburg.de/~cziegler/BX/). Once again, the tests are run with a training set of 10.000, and a test set of 90.000. This training dataset has the most items: 6.327. Also, unlike the other two datasets, this one does not have timestamps, meaning they can only be processed in random order. That is expected to greatly decrease effectiveness.

Model	*Training time (s)*
Item-to-Item with Jaccard	90
Euclidean Item Recommender	11169

The results are very similar to what was seen at Netflix, except, they are even more visible.

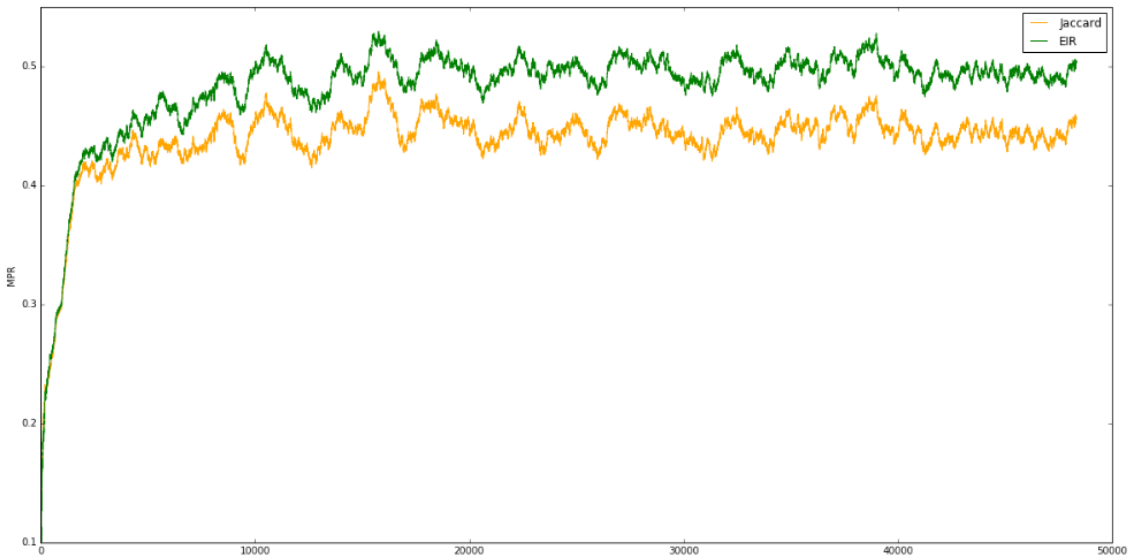

Fig. 9.: Moving average of MPR of the two models (GoodReads)

The MPR starts off low, and they are almost at the same level. However, they rise very fast, and just like earlier, *EIR* rises faster. The number of successful recommendations is even lower, but it is still at an acceptable level.

Based on the tests, it appears that the less complex *Item-to-Item with Jaccard similarity* is the superior choice against *Euclidean Item Recommender* in every aspect. Not only it is significantly faster, but also gave better results on all 3 datasets.

8 Combined model

Evaluating the results of the simulation, it is visible there is no clear “best” model. However, it might be possible to predict which model will give the best results for a given user. The idea is that for each user, keep track of how each model performed for the last *n* requests. This can be done for different measurements, in this case, RMSE, DCG, and recall are used.

The metamodel contains multiple dictionaries for storing user data. The key is always the user’s id, and the value is another dictionary. This inner dictionary has and id for each model as a key, and a fix sized array as a value.

Whenever a user rates an item, the data is also forwarded to the combined model. The RMSE is saved every time, while the rank is only saved, if the rating was 4, or better. These are the ratings that the metamodel considers “liked” by the users. Storing the rank of items that the user did not like would make the results worse. These items are expected to be at the end of the ranking. If the combined model would store these, and would treat it the same as the liked items, then it would conclude that the model it is looking at ranked an item that a user liked low. The reality on the other hand is that the item has a low rank, because the model knew, that the user wouldn’t like it. Both DCG and recall can be calculated from the stored ranks.

It is possible, that the metamodel cannot give a model recommendation yet. This happens every time a user does something for the first time. But with recall and DCG, it might happen a lot more often, since it might take some tries to find an item that the user rates as 4 or better. For these cases, the default recommended model is *MF with ALS*.

The performance overhead of the metamodel should also be noted. During the simulation, that took more than 13 hours, the calculations of the metamodel only took 402 seconds. This is less than 1% of the total simulation time. This is in line with the expectations: the combined model’s overhead is linear in the number of models it keeps track of.

While not part of the combined model, only the individual models, it is also worth mentioning the strategy for when prediction is not possible, due to unknown user, or element. This happens a lot: the percentage of successful predictions is below 35%. The

reason for failed prediction is almost always an unknown user. In such casees, the strategy for giving prediction for one user-item pair is predicting a rating of 3.

From the users' perspective, it is more important what the recommendation is in such case. With the matrix factorization models, when the user is unknown, items are ranked by their average predicted ratings. However, this is not acceptable for the k-Nearest-Neighbors model: doing so requires knowing the full prediction matrix. The problem with that is the same as with the item-distance matrix: calculating it takes a very long time, and it this case, a large amount of recalculation with every update is not avoidable. Because of this, *kNN* is not able to give a recommendation if the user in question is not known by the model.

Picking different measurements for optimization will yield different results. When observing the results for each measurement, always the best combined model will be used.

In each case, the moving average of the last 1.000 elements are shown on the plots. For the first 1.000 elements, this is the average of all elements, and after that, the average of the last 1.000 elements.

8.1 RMSE

Looking at the plot for RMSE, it seems that staying below the minimum of the 3 models with the combined (optimized for RMSE) model was a success:

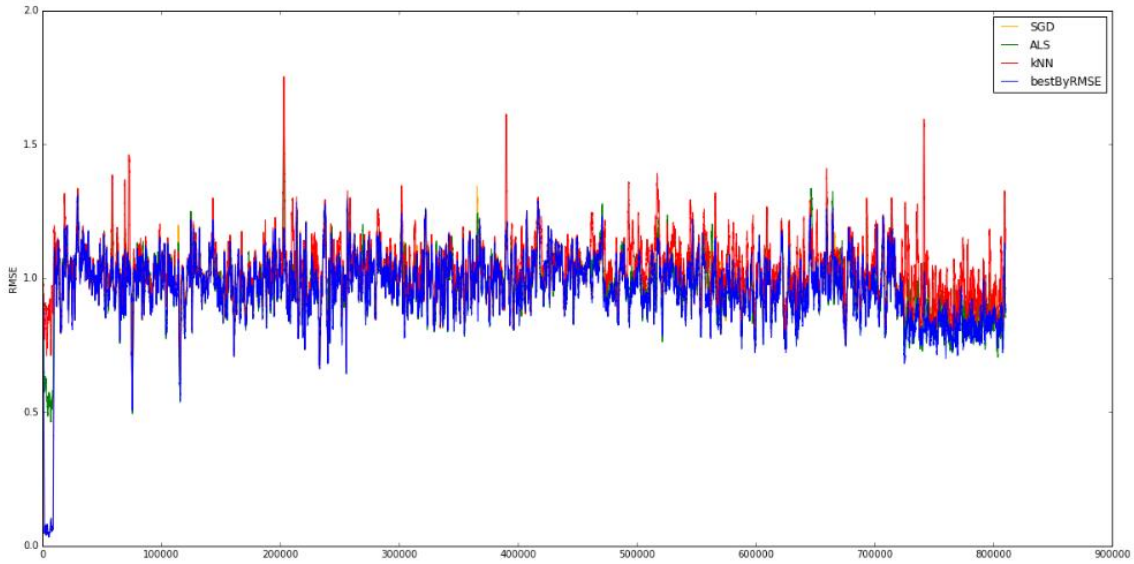

Fig. 10.: Moving average of RMSE for each model

To be sure, look at the difference with each model. The averages should be below zero.

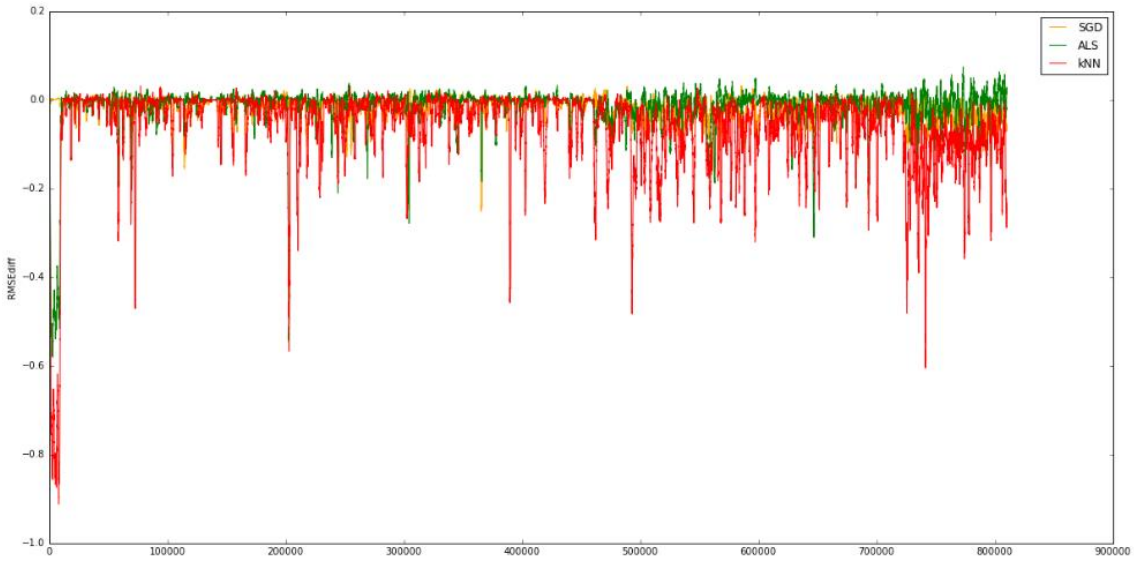

Fig. 11.: The difference of RMSE of the combined model from the base models

On average, the combined model managed to stay below the RMSE of each base model.

8.2 Recall

As before, when evaluating recall, only the elements that have a rating of 4 or above are considered.

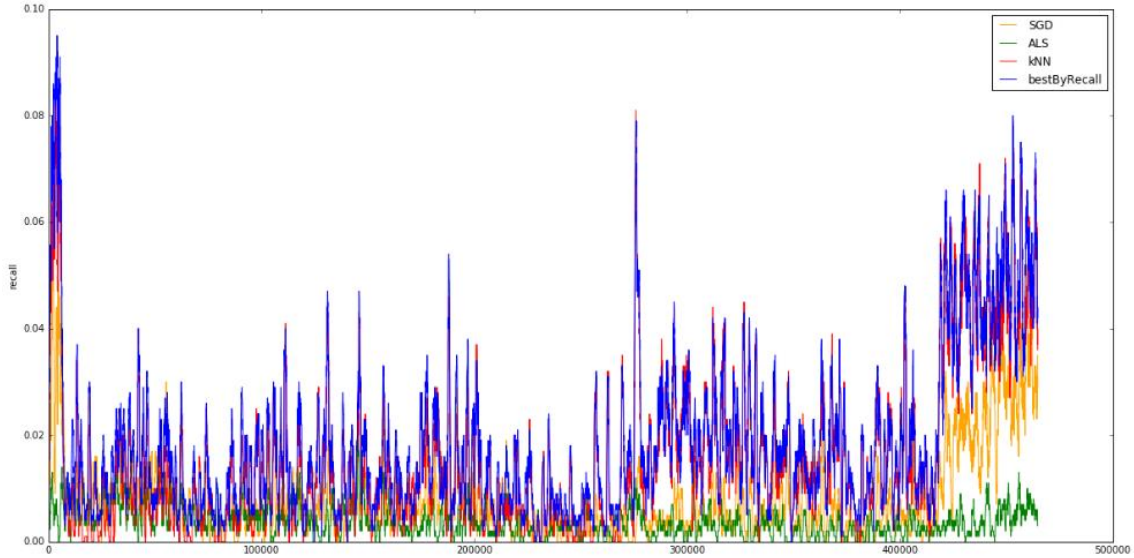

Fig. 12.: Moving average of recall for each model

It appears that the model optimized for recall also succeeded, the moving average of the recall for the combined model is constantly above the other two models. Looking at the differences also suggests the same.

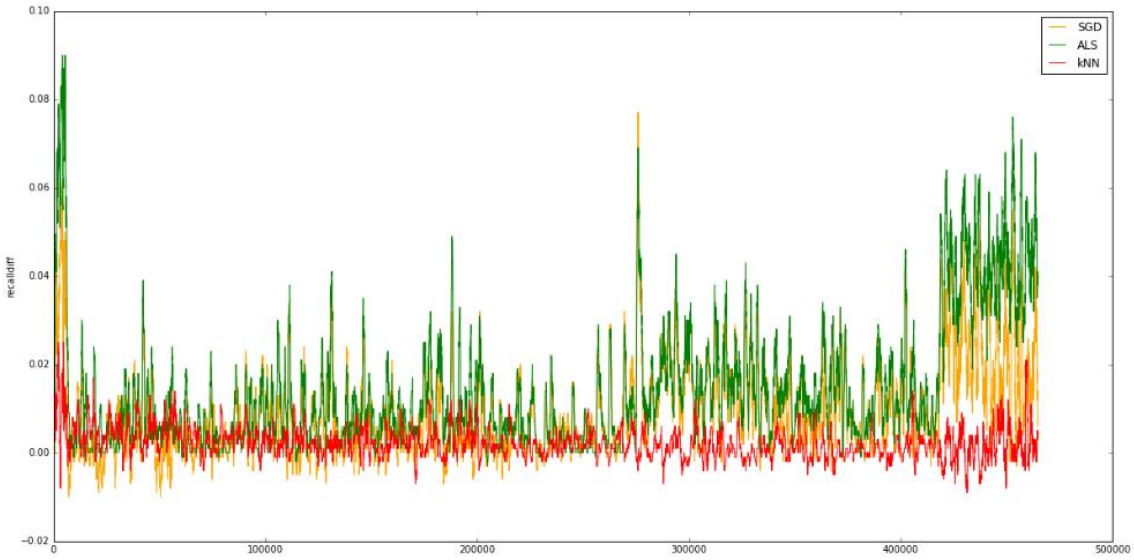

Fig. 13.: The difference of recall of the combined model from the base models

It is visible, that the numbers are overwhelmingly positive, as expected.

8.3 DCG

As usual, only elements with a rating of 4 or higher are considered. The results should not come as much of a surprise: the DCG of the combined model is constantly above the DCG of the other models.

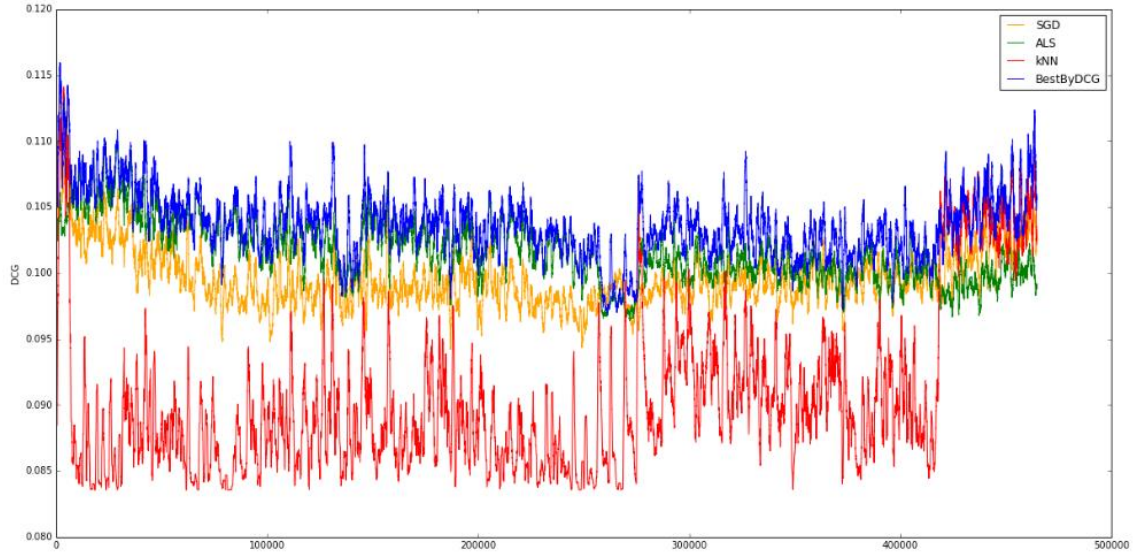

Fig. 14.: Moving average of DCG for each model

The differences are also above zero, just as expected.

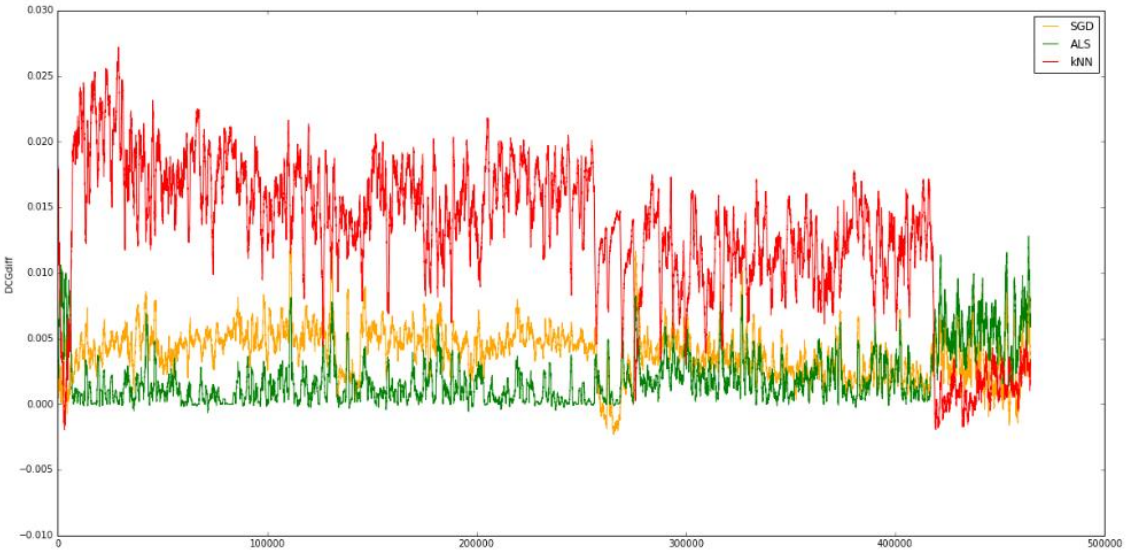

Fig. 15.: The difference of DCG of the combined model from the base models

9 Scheduling updates

The performance of the models is also affected by retraining strategy. For the following measurements, the initial training dataset was 20,000 elements. After that, 100,000 requests were sent, with different update strategies. The strategies were always frequency based. The goal was to see how performance changes if during the 100,000 requests 0, 1, 3, or 7 updates are executed.

Number of updates	*Time (s)*
0	9433
1	5045
3	3401
7	2811

Notice how by increasing the frequency of updates, the overall time might decrease. The reason for this is simple: the updater stores the messages that will be given to the model at the next update. Having to work with a collection with too many elements causes a visible drop in performance. By increasing the frequency of updates, the time will not always decrease. The more frequent the updates are, the more the overall execution time will be affected by the actual update time, and less by the performance of the update proposer. E.g.: if the basis for *kNN* was the ratings, then 7 updates would probably take much longer than 0 updates.

The next measurement to observe is the rate of successful predictions.

Number of updates	*Successful predictions (%)*
0	1.9
1	5.8
3	9.5
7	12.9

Here, the results are as expected: more frequent updates cause a higher percentage of successful predictions. However, the following measurements will show that this doesn't necessarily lead to better results, especially if there aren't enough ratings for a

new user or item. In this case, recommending the global top items is probably better than giving personalized recommendation.

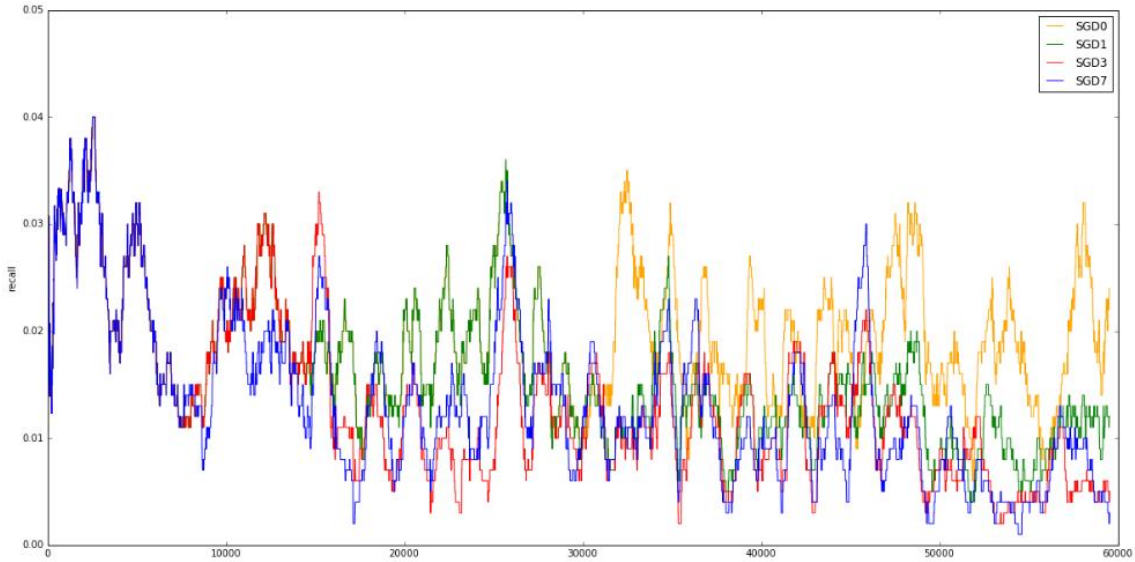

Fig. 16.: More successful recommendations don't necessarily mean better recall (MF with SGD)

Recall visibly dropped after every single update. By the second half, the model without updates dominates the rest. For DCG, the trends are very similar.

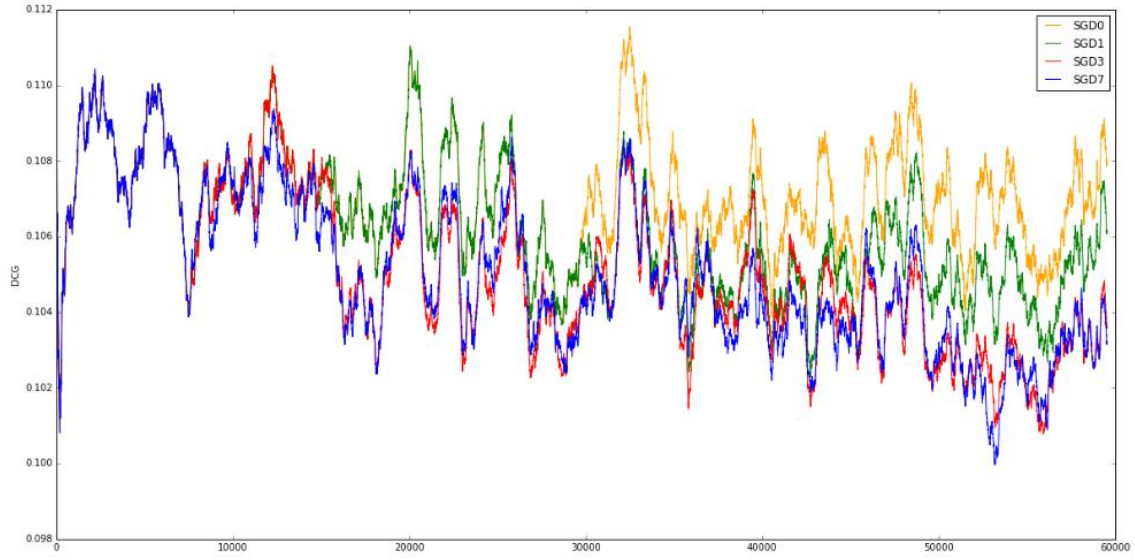

Fig. 17.: The data for DCG is in line with the data for recall (MF with SGD)

Perhaps on this plot, it is even more visible. The models give the same results before updates, but after an update happens, the models that come out with better results are the ones that were not updated.

Finally, the average RMSE does not seem to be affected much during the simulation.

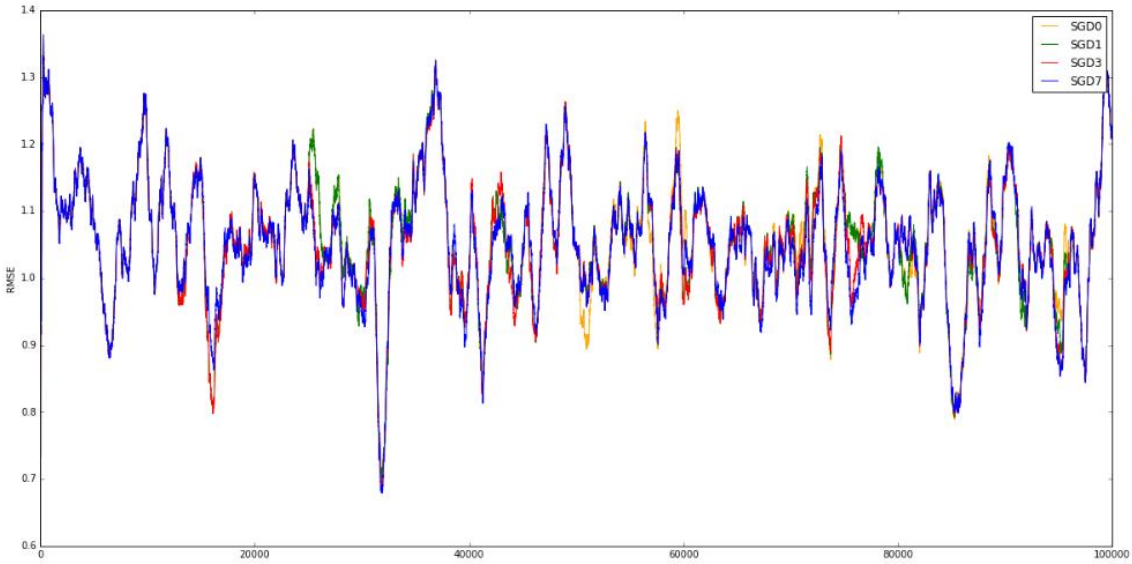

Fig. 18.: RMSE does not seem to be affected during the first 100k messages (MF with SGD)

The plots are essentially the same for the other models too, so they are not included here.

Based on this, the conclusion is the following. More frequent updates can cause an increased overall performance, and will lead to a better rate of successful predictions. However, unless there is sufficient data available for each user/item, this may lead to worse metrics. Because of this, it is essential to set a minimum limit for new users/items to be added to the model. To illustrate this, here is a comparison of the recall metrics with a 5^{th} strategy, that includes 7 updates, but only adds a user or item, if it has at least 5 ratings that belong to it. The reason the 7-update strategy was chosen is because that was the strategy that gave the worst overall measurements in the previous tests.

Execution took 2643 seconds, with a success rate of 12.1%.

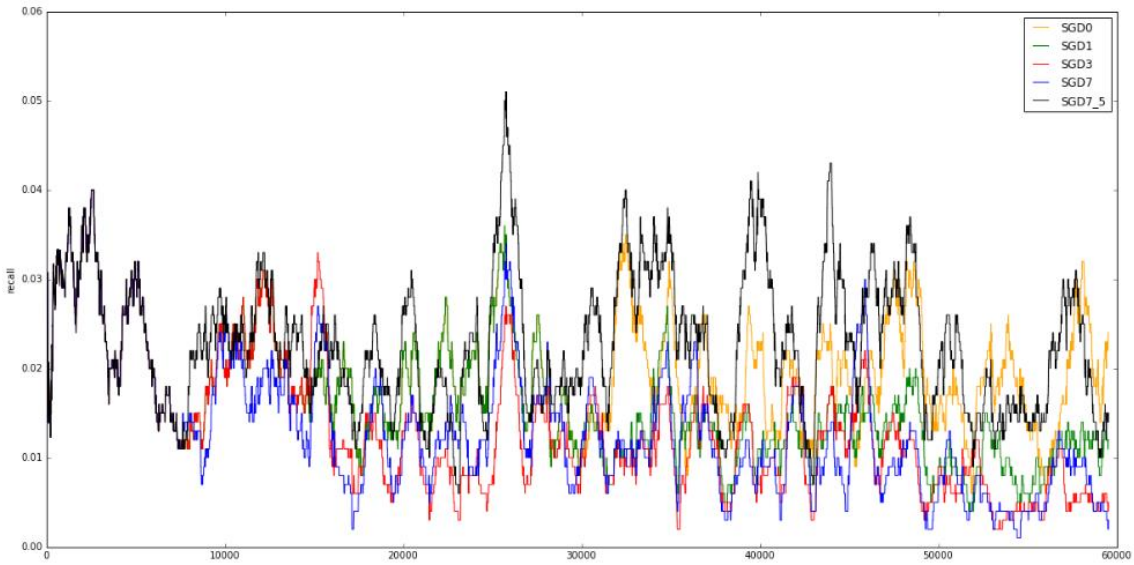

Fig. 19.: Setting a minimum rating limit for users/items can lead to better results

While there is still room for improvement, it is visible how much difference does it make to set a minimum required number of ratings, even if the limit is very low. Now, most of the time the new strategy dominates the others.

However, it is also worth considering, that many users might simply not care about personalized recommendations, and will just go with whatever is popular. It could also explain why a higher rate of unsuccessful personalized recommendations can lead to better metrics. For these users, the global recommendation might be a better choice, then the personalized one. The combined model introduced in chapter 8 can give a very good solution for this. If the global recommendations are treated as separate cases, then the combined model can learn which users tend to lean more towards the overall more popular items.

To best way to see if that this is the case, is to run another simulation on the same dataset. As the last time, 7 updates are executed to maximize the ratio of successful predictions, and a 5 ratings limit is set on new users and items to ensure more personalized recommendations. The simulation uses the two Matrix Factorization models, but this time they also return the global recommendation along with the personalized recommendation every time. Once again, the measurement is recall.

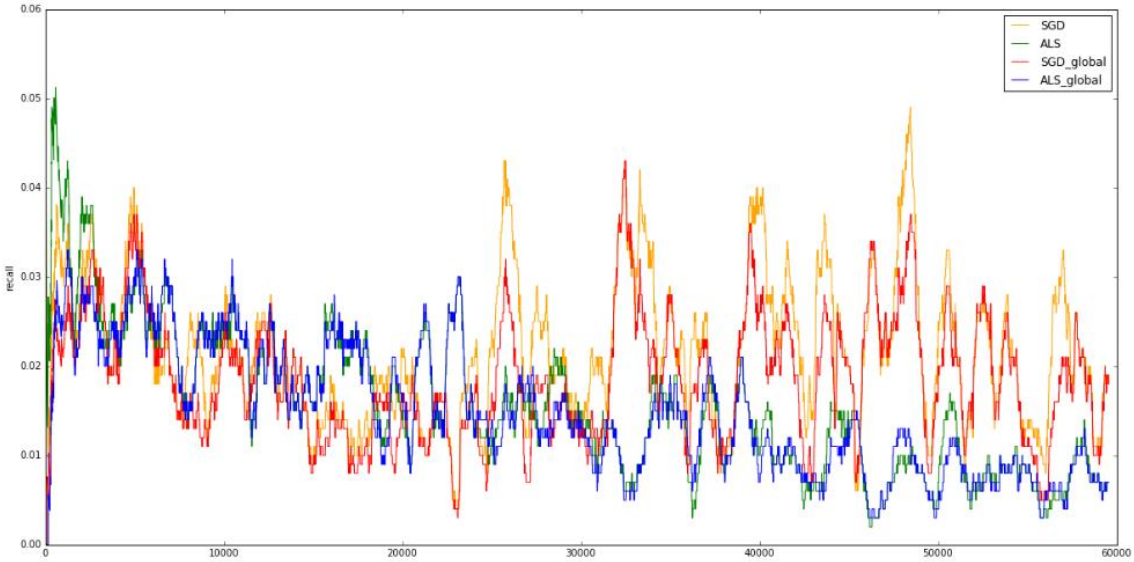

Fig. 20.: Recall of personalized and global Matrix Factorization recommendations

The results suggest that this assumption might be true. While there is no single model that constantly dominates the others, it appears that one of the global recommendations have the best recall averages relatively often.

10 System-wide performance considerations

There are several performance concerns, that should be measured through the whole system. From the users' perspective, one of the most important questions is "How fast can my request be served?". Recommendations should be fast, even when there are background tasks running (such as updating models).

Processing new information should be asynchronous. Updating the models must never make the system unresponsive, the user shouldn't notice anything from the update running. These issues are addressed in this section.

10.1 Processing messages

In an earlier version of the recommender, new ratings simply came through the message queue, and were processed one by one. When an update was triggered, processing messages from the queue stopped, until the update was finished. This wasn't a serious issue, until some update times became very long. So long, that RabbitMQ closed the connection with the recommender, because it was not responding. It was understandable too, at that time, ratings as a basis for *kNN* was still under consideration. Updating that model often took more than an hour.

Because of that, message processing was rewritten in a way, that the receiver fetches the messages from RabbitMQ, then schedules the actual processing asynchronously. This way, the model can still receive requests while update is happening. The next test involves sending 10,000 requests to the message queue. As seen on RabbitMQ's management surface, messages are processed almost immediately.

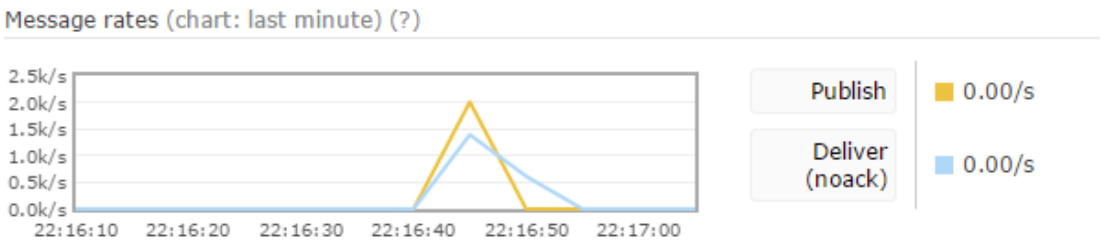

Fig. 21.: Messages are processed almost immediately

For the recommender, processing these messages takes a lot longer, especially since in this case the updater was set to start an update every 1.000 messages. That does not concern the message queue though.

CPU cores are only used on 100%, when the recommender fetches many messages from the queue. Otherwise, even when an update is in progress, the usage decreases.

10.2 Giving recommendations

Giving a pre-calculated recommendation to a user is very fast: the java application reads it from the database, and returns it. The time necessary for pre-calculating the recommendations however, can vary between models.

For *kNN*, it is very slow. It would require calculating the whole prediction matrix. This is the same problems as using the ratings as a basis for the model. However, if doing that is necessary, the best solution is probably calculating recommendations one-by-one, when they are needed. Most of the users won't request recommendations between updates anyway, so a lot of unnecessary calculation can be avoided. Although if this strategy is applied, then it is not possible to give a global top recommendation list based on item averages, since it would also require calculating the whole matrix first. Furthermore, even if calculating the prediction matrix was fast, it would take a lot of memory. In case of a recommender, that works with millions of songs, it would not be possible to store it in-memory.

Matrix factorization models perform a lot better. For a model of 176 users and 2401 items, giving a t*op 100* recommendation for every user (+plus a global top list) only took *0.6 seconds*. After that, saving all the recommendations into the database took an extra *0.14 seconds*. This is more than acceptable, the increase in the training times (see chapter 7) are not even noticeable.

11 Conclusions

As it is visible from the simulations, there is still a long way to go when it comes to researching recommendation systems. Each user's behavior can be so different, that it is not likely that a single model will ever be able predict what every user will be interested in the future. This is probably the reason why YouTube is also giving each user multiple sets of "recommendations". The homepage contains multiple sets of videos, that are recommended based on the user's history, sets that are compiled of videos that are uploaded by people, that the user explicitly said he/she is interested in, and another set, that only contains videos, that are popular at the time. Different people visit the site for different reasons, and most of them probably only care about some of these recommendation sets. Because of this, metamodels, such as the combined model introduced in chapter 8 can be very useful: if the recommendation system can tell which recommendation set matches the behavior of each user the best, user experience can grow a lot. Of course, using metamodels always adds some overhead. In this case, that was not much, but not all metamodel overheads are linear in the number of models it works with. For example, take a metamodel that determines predictions by using a personalized linear combination of the predictions given by the models. It would have to calculate the elements of the prediction matrix once again to give recommendations.

After experimenting with multiple models, the ones I would choose to keep are the ones using Matrix Factorization, and the Item-to-Item recommender, that uses Jaccard similarity. The latter is self-explanatory based on the results: it is significantly faster, and gave better results on every single dataset. With Matrix Factorization, the main deciding point is also speed: these algorithms are a lot more efficient than k-Nearest-Neighbors. With very large datasets, I would go with the version the uses Stochastic Gradient Descent, since it scales better, but in many cases, Alternating Least Squares can also give good results in an acceptable timeframe.

I managed to create a recommendation system, that can efficiently give recommendations for both users and items, and that is ready to be used by a separate application. It can also handle cases, when a visitor has no history available to the system. The models can give global recommendations, and as measurements suggest, for many

users, these are even better than personalized recommendations. They also have access to all the item-to-item recommendations, since those don't require any user history.

As it was seen, the only input the recommendation system requires is the messages that give information about user activity, and all the recommendations are put in a database. This means that not only any application that has access to the database can get the recommendations, but also that their performances are not affected by the update schedules of the recommendation system.

In such an active area of research, there are always more new things to try. New algorithms are developed all the time. The metamodel seemed very promising, different variations could be tried for that. By using the session's history of seen items, item-to-item recommenders could also be included in it. And even for the already implemented algorithms, many improvements and measurements could be done. A technique often used to avoid having to calculate large matrices, is random sampling. Using that, some of the slower algorithms could be sped up.

The thesis also did not address the problem of overfitting – a problem that can reduce the predictive power of the models. Overfitting occurs, when the model tries to fit too closely to the training data. The problem is that the training data is usually not perfectly representative of reality. The ratings can be influenced by many outside factors, that are not directly observable on the data – such as the mood of the user at the time of rating, or a rating that is mistakenly set at the wrong value. If the model fits so close, that it also reproduces these inaccuracies, then the predictions might also have such errors in them.

It is certain, that with a constant search for improving user experience, the importance of recommendation systems is going to grow even more in the future. While in this case, ratings were predicted, these techniques could have other uses. For example, by storing students' test results, their performance could be predicted on tests, that they haven't taken yet. These results could then be used for recommending books or courses, that they would benefit from the most.

Considering all these, in my opinion, researching recommendation systems is worth investing time, money, and energy into.

References

[1] Koren, Y., Bell, R., & Volinsky, C. (2009). *Matrix factorization techniques for recommender systems*. Computer, (8), 30-37.

[2] Said, A., Dooms, S., Loni, B., & Tikk, D. (2014, October). *Recommender systems challenge 2014*. In Proceedings of the 8th ACM Conference on Recommender systems (pp. 387-388). ACM.

[3] Rendle, S., & Freudenthaler, C. (2014, February). *Improving pairwise learning for item recommendation from implicit feedback*. In Proceedings of the 7th ACM international conference on Web search and data mining (pp. 273-282). ACM.

[4] Adomavicius, G., & Tuzhilin, A. (2011). *Context-aware recommender systems*. In Recommender systems handbook (pp. 217-253). Springer US.

[5] Ma, H., Zhou, D., Liu, C., Lyu, M. R., & King, I. (2011, February). *Recommender systems with social regularization*. In Proceedings of the fourth ACM international conference on Web search and data mining (pp. 287-296). ACM.

[6] Chaney, A. J., Gartrell, M., Hofman, J. M., Guiver, J., Koenigstein, N., Kohli, P., & Paquet, U. (2014, June). *A large-scale exploration of group viewing patterns*. In Proceedings of the 2014 ACM international conference on Interactive experiences for TV and online video (pp. 31-38). ACM.

[7] Koenigstein, N., & Koren, Y. (2013, October). *Towards scalable and accurate item-oriented recommendations*. In Proceedings of the 7th ACM conference on Recommender systems (pp. 419-422). ACM.

[8] Netflix Prize, http://www.netflixprize.com/ (27.10.2016.)

[9] Yining Wang, Liwei Wang, Yuanzhi Li, Di He, Wei Chen, Tie-Yan Liu. 2013. A *Theoretical Analysis of NDCG Ranking Measures*. In Proceedings of the 26th Annual Conference on Learning Theory (COLT 2013).

[10] Roy Thomas Fielding, 2000. *Architectural Styles and the Design of Network-based Software Architectures*. (ch. 5)

Appendix

Models

Parameter name	*Description*	*Value*
l	Learning rate of the Stochastic Gradient Descent	0.03
c	The range the values the initial random number can come from	0.5
d	The common dimension of U and V	100
runPerRound	The number of times each known value is visited during the training	10

Table 1.: Parametrization of "Matrix Factorization with Stochastic Gradient Descent" model

Parameter name	*Description*	*Value*
c	The range the values the initial random number can come from	0.5
d	The common dimension of U and V	10
runPerRound	The number of times each known value is visited during the training	3

Table 2.: Parametrization of "Matrix Factorization Alternating Least Squares" model

Parameter name	*Description*	*Value*
k	The number of nearest neighbors the algorithm takes into consideration	10

Table 3.: Parametrization of "k-Nearest-Neighbors" model

Parameter name	*Description*	*Value*
l	Learning rate of the Stochastic Gradient Ascent	0.03
d	Size of the item vectors	10

Table 4.: Parametrization of "Euclidean Item Recommender" model

Update Proposer Module

Parameter name	*Description*	*Value*
recallStored	The number of messages the moving average of recalls is calculated from	100
dcgStored	The number of messages the moving average of dcgs is calculated from	100
messageCountLimit	Update is always proposed after this many messages	25000
minMessagesToUpdate	The minimum number of incoming messages necessary since the last update for an update to be proposed	10000
recallLimit	Update proposed, if the moving average of recalls is below this limit	0.33
dcgLimit	Update proposed, if the moving average of dcgs is below this limit	0.125
unknownLimit	Update proposed, if the rate of requests where the model couldn't give prediction is above this number	0.7
newItemCountLimit	Minimum number of ratings necessary for a new item to be put in the model	5
newUserCountLimit	Minimum number of ratings necessary for a new user to be put in the model	5

Table 4.: Parametrization of the Update Proposer Module

Interfaces

Message queue message format

```
{
  "contentId": {
    "type": "string"
  },
  "ownerId": {
    "type": "string"
  },
  "rating": {
    "type": "integer"
  }
}
```

Database rating storage format

```
{
  "contentId": {
    "type": "string"
  },
  "ownerId": {
    "type": "string"
  },
  "rating": {
    "type": "integer"
  },
  "timestamp": {
    "type": "string",
    "format": "date-time"
  }
}
```

Database recommendation storage format

```
{
  "type": "object",
  "properties": {
    "key": {
      "type": "string",
      "description": "id of the user the recommendation belongs to, or
'top' for the recommendation of the overall best items"
    },
    "value": {
      "type": "array",
      "elements": {
        "type": "string"
      },
      "description": "a list of the ids of the items recommended for the
user"
    }
  }
}
```

Movies

Field name	*Description*
id	integer, it serves as a unique identification for the movie. While it may look sequential at first, it is not
title	string, the title of the movie
year	integer, the year the movie came out
genres	array of strings, the genres the movie belongs to. It can be one, or more of the following: *Action, Adventure, Animation, Children's, Comedy, Crime, Documentary, Drama, Fantasy, Film-Noir, Horror, Musical, Mystery, Romance, Sci-Fi, Thriller, War, Western*

Table 5.: The "movies" dataset

Ratings

Field name	*Description*
userId	integer, the unique identifier of the user the rating belongs to
movieId	integer, the unique identifier of the movie the rating belongs to
rating	integer, the value of the rating on a 1-5-star scale (whole-star ratings only)
timestamp	integer, represented in seconds

Table 9.: The "ratings" dataset