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

Laki Dániel

SESSION-BASED IMPLICIT AND EXPLICIT RECOMMENDATION

DEPARTMENTAL CONSULTANT

Simon Gábor

EXTERNAL CONSULTANT

Daróczy Bálint

BUDAPEST, 2016

Table of contents

Összefoglaló ... **4**

Abstract ... **5**

1 Introduction ... **6**

2 Models ... **8**

2.1 Matrix Factorization with Stochastic Gradient Descent ... 8

2.1.1 Preprocessing ... 8

2.1.2 Matrix factorization ... 8

2.1.3 Initialization ... 9

2.1.4 Choosing the next value ... 10

2.1.5 Measurements ... 10

2.1.6 Stochastic Gradient Descent ... 12

2.2 Matrix Factorization with Alternating Least Squares ... 13

2.2.1 Alternating Least Squares ... 14

2.3 k-Nearest-Neighbors ... 15

2.3.1 Item distances ... 16

2.3.2 Prediction ... 16

3 Updating the models ... **18**

3.1 Update Proposer Rules ... 18

3.1.1 Frequency-based rules ... 18

3.1.2 Measurement-based rules ... 19

3.1.3 Success-based rule ... 19

3.1.4 Rules about new items ... 20

4 Architecture ... **21**

4.1 Recommender ... 21

4.1.1 Framework ... 21

4.2 System-wide architecture ... 24

4.2.1 Client ... 24

4.2.2 Java application ... 24

5 The simulation ... **26**

6 Combined model ... **29**

6.1 RMSE ... 29

6.2 Recall ... 30
6.3 DCG ... 32
7 Scheduling updates ... 33
8 System-wide performance considerations ... 37
8.1 Processing messages ... 37
8.2 Giving recommendations ... 38
9 Conclusions ... 39
References ... 40
Appendix ... 41
Models ... 41
Update Proposer Module ... 42
Interfaces ... 43
Message queue message format ... 43
Database rating storage format ... 43
Database recommendation storage format ... 43
Movies ... 44
Users ... 45
Ratings ... 47

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]. Dolgozatomban összehasonlítok több ismert ajánlórendszert és bemutatok egy, több modell kombinációjának segítségével, session-ök esetében is alkalmazható modellt.

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]. In my essay, I compare multiple well-known recommender systems, and introduce a model that is a combination of multiple models, and can be used in case of sessions.

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].

These systems have different qualities, and there isn't necessarily a "best" one. Their performance can be evaluated from a lot of aspects, and in different situations, different recommender systems might be better. It is even possible, that some recommenders fit the behavior of individual users better.

This last assumption serves as a basis for this essay. The goal is to create a combined system, that can monitor, and evaluate the performance of other recommender systems of different qualities. The combined system doesn't even have to know how the other models work, the only important information about them is the prediction, and recommendation they give. Ideally, based on these, the combined system should be able to tell, that at any given time, for any given user, which of the known recommender systems will give the best results for the user.

In the first part of the essay, 3 well-known recommendation models will be introduced. They have different qualities; these will be inspected in the essay. Some methods of evaluating the performance of these models will also be introduced. After that, the essay will show strategies for updating the models.

The combined system is developed for a larger system used in the industry, that relies heavily on recommendations. Parts of this system, that is related to the recommender are also introduced here.

Finally, the essay tries to answer important questions in 3 areas:

1. Can such a combined system work? Can it give the users better recommendations, then what they would get from the base models?

2. How does updating affect these models? What is a good strategy for running updates on the models?
3. How does such system perform under heavy load? Is it even possible, to effectively run such a system?

2 Models

This section contains a description of three well-known models that are often used for recommender 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 model 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 essentially 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 do we 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 we 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.

2.1.6 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:

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

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

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 even make the prediction 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:

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

Where k is the common dimension of U and V.

After rearranging it:

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

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 seriously 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.

To greatly reduce retraining times, the genres of the movies were used as a basis in the current model. 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.

3 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.

3.1 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.

3.1.1 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.

3.1.2 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.

3.1.3 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.

3.1.4 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 don't have enough ratings belonging to them, 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.

4 Architecture

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

4.1 Recommender

The recommender 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.

4.1.1 Framework

Many of the processes in the recommender 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.

4.1.1.1 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.

4.1.1.2 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 data (see chapter **Hiba! A hivatkozási forrás nem található.**) 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 currently only used for testing. Receiving messages however is very important. This is how the recommender learns about new ratings.

4.2 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.

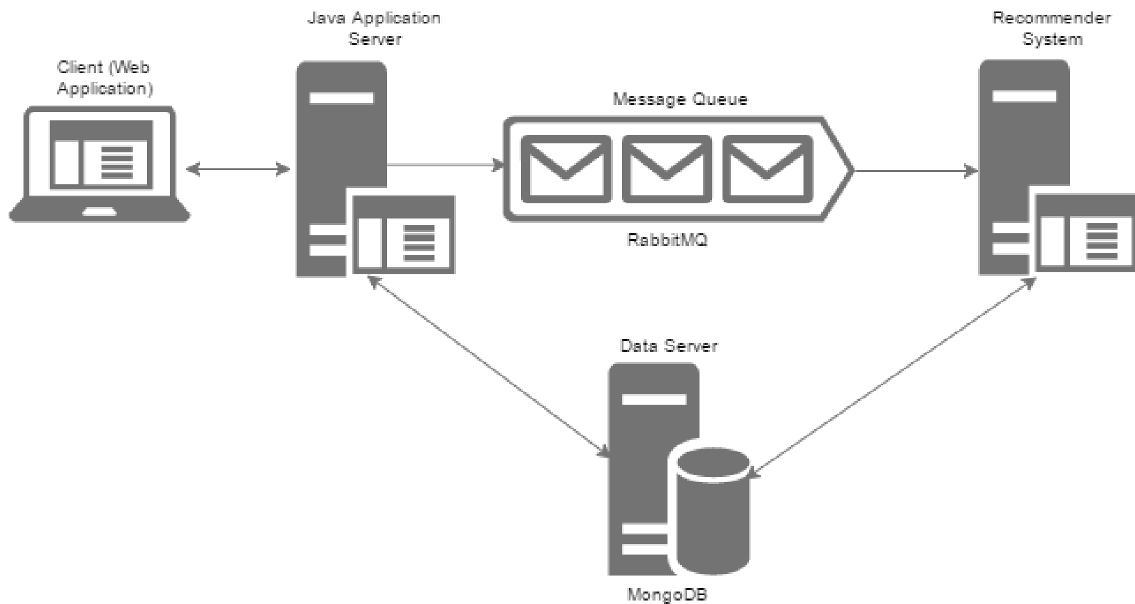

Fig. 1: System-wide architecture

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.

4.2.1 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.

4.2.2 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'.

5 The simulation

To test how the recommendation engine performs, some preexisting data is necessary. For this purpose, my 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.

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

This section contains some basic measurements regarding performance during these two periods. Evaluating the results however is not part of this section. That can be found in chapter the *Combined model*.

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.

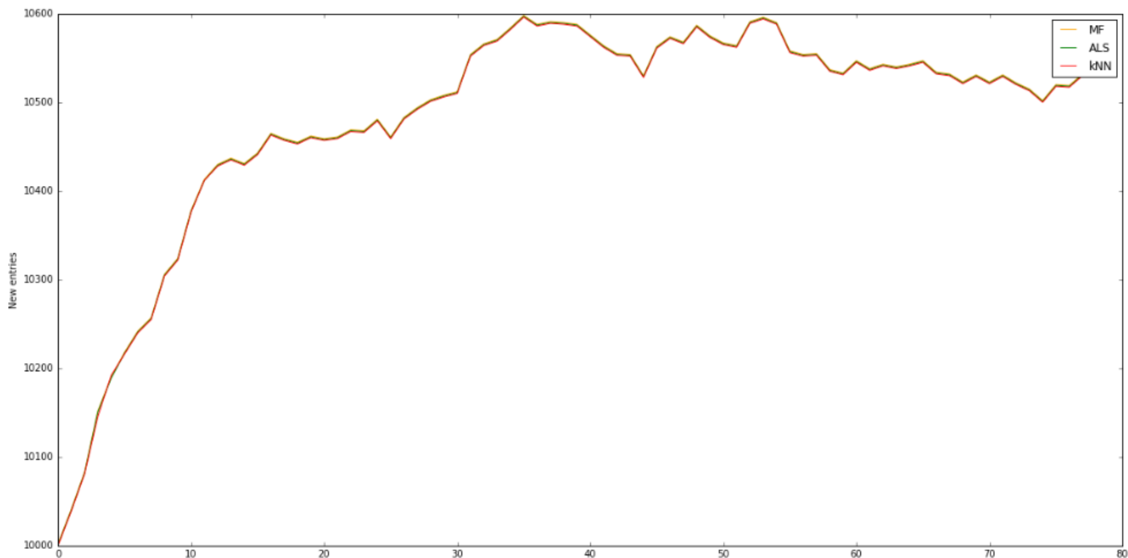

Fig. 2.: Number of new ratings put in the models after each update

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.

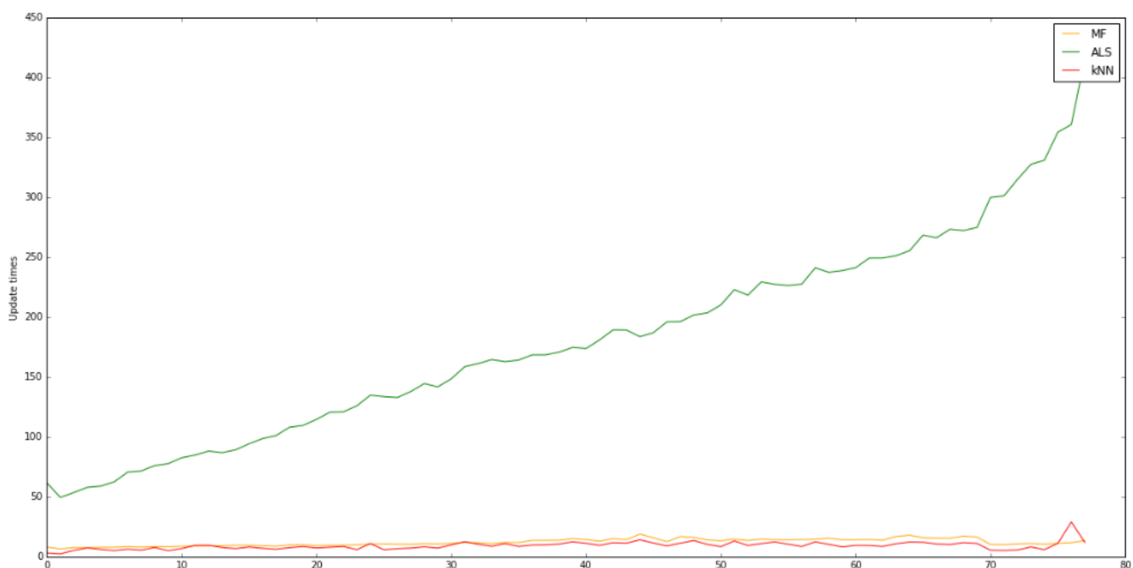

Fig. 3.: Update times (s) for each model.

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.

6 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.

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 case, 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 recalculating 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.

6.1 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:

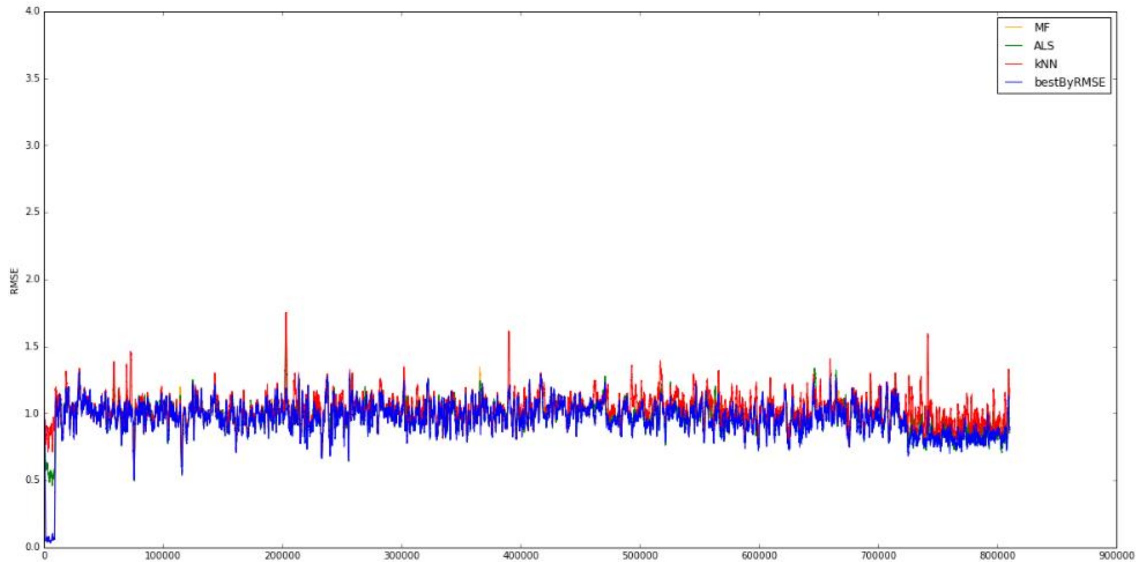

Fig. 4.: Moving average of RMSE for each model

To be sure, look at the difference with each model:

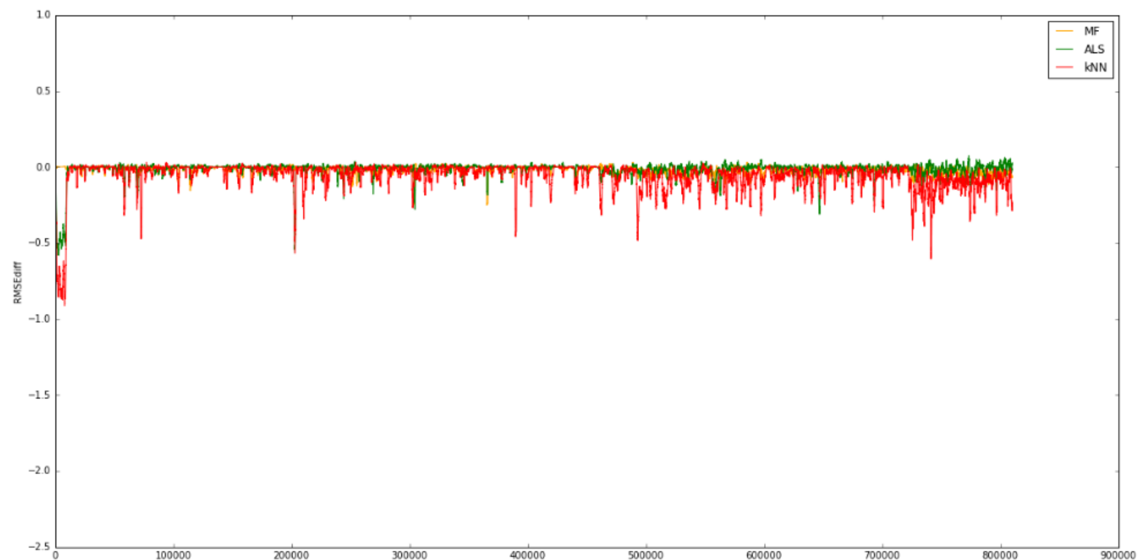

Fig. 5.: The difference of RMSE of the combined model from the base models

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

6.2 Recall

The way recall should be evaluated is not trivial. For each single element, they are 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, the next measurements are evaluated only for items, that got at least a rating of 4 from the user. Also, since *kNN* cannot give recommendations for unknown users, it is left out from here.

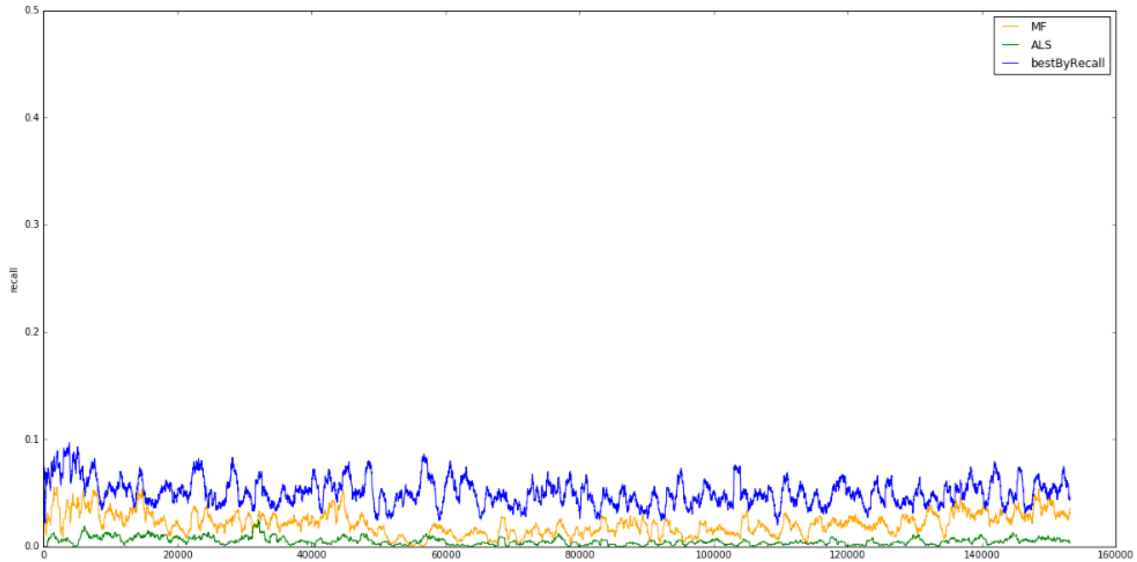

Fig. 6.: Moving average of recall for each model

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.

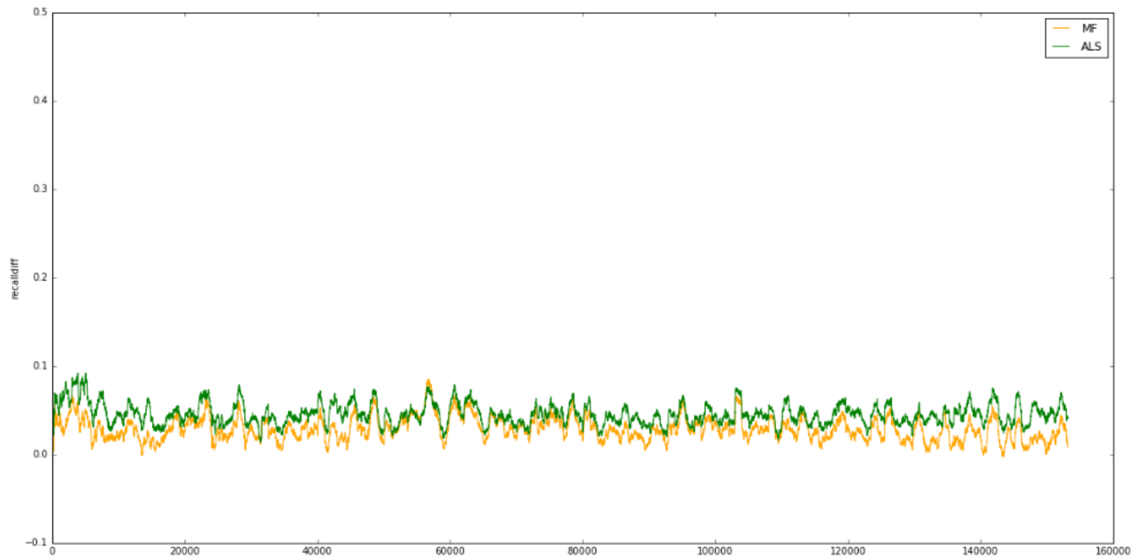

Fig. 7.: The difference of recall of the combined model from the base models

It is visible, that the numbers are overwhelmingly positive.

6.3 DCG

While the combined model for RMSE and recall seem to yield results immediately, apparently DCG requires some time before it starts working as intended.

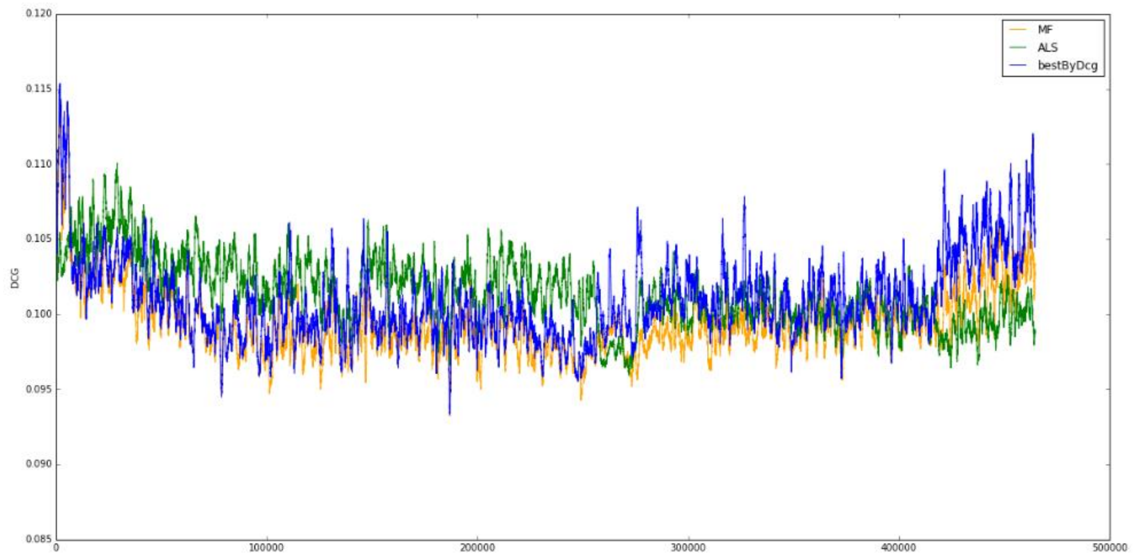

Fig. 8.: Moving average of DCG for each model

In the first half, *MF with ALS* performs better, but by the second half, it overtakes it, and starts performing better, that the base models. The difference-plot suggests the same.

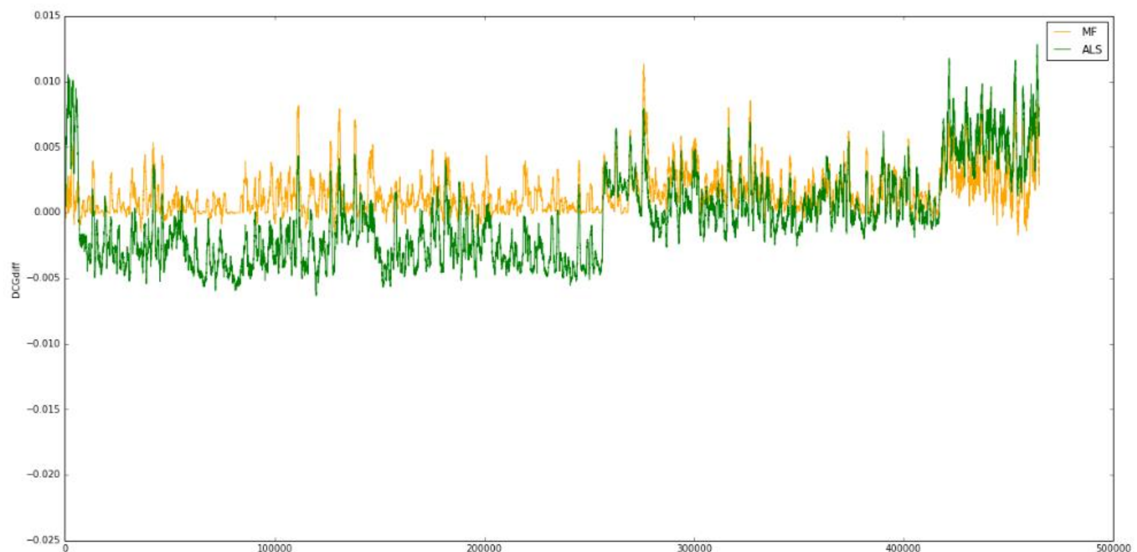

Fig. 9.: The difference of DCG of the combined model from the base models

7 Scheduling updates

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. This is especially true, 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.

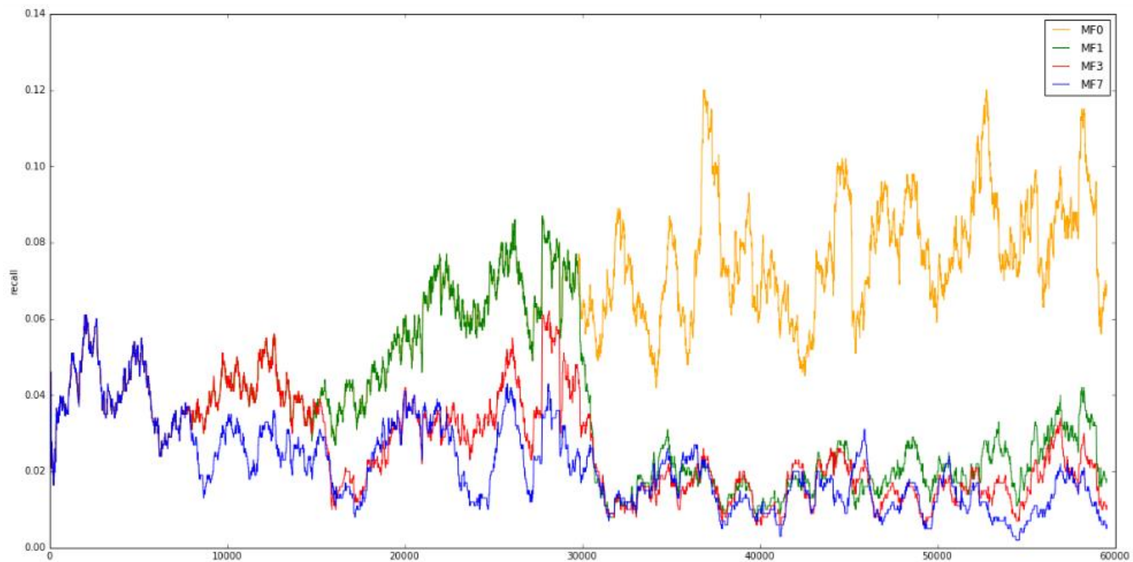

Fig. 10.: Recall might decrease with more personalized recommendations (MF with SGD)

The plot for *MF with ALS* looks essentially the same, so it is not included here. For DCG, different trends can be observed for *MF with SGD*.

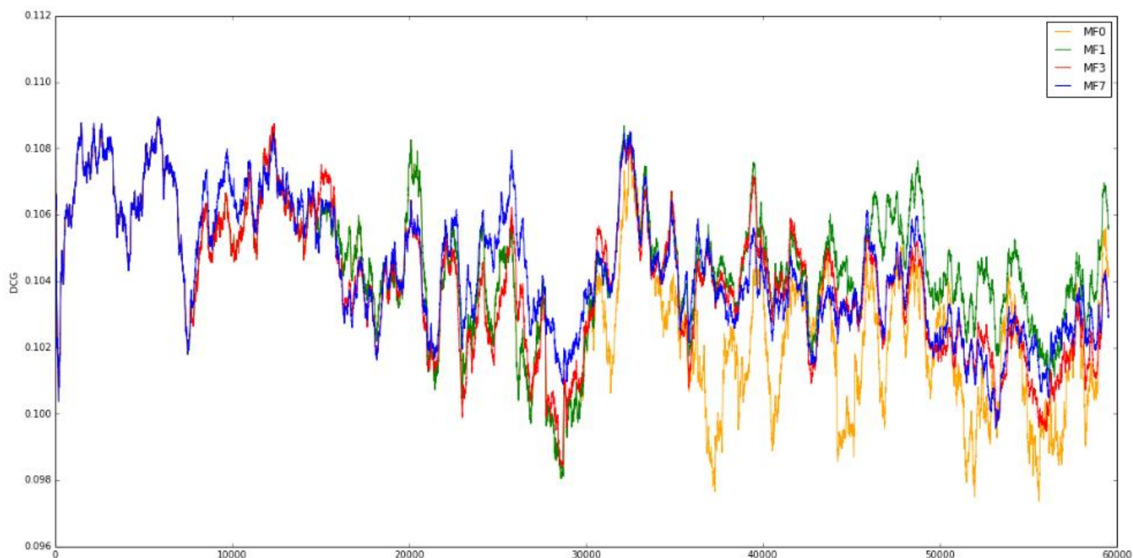

Fig. 11.: Without update, DCG got worse over time (MF with SGD)

On the other hand, while the same trend can be observed for *SGD* and *ALS* for recall, for DCG, *ALS* is more in line with what the recall data suggests.

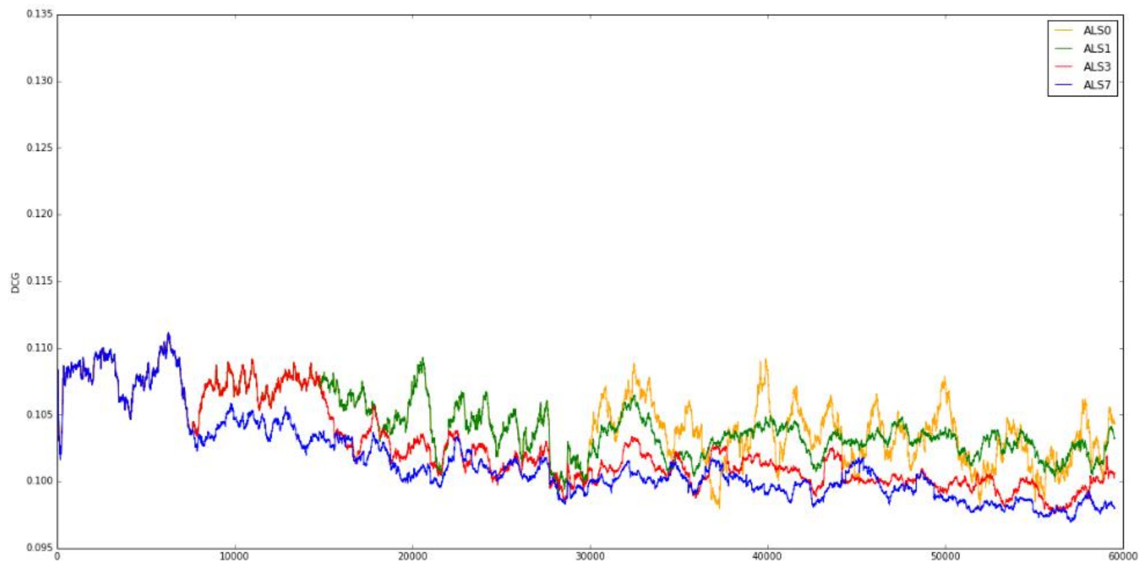

Fig. 12.: DCG for MF with ALS is more in line with the data from recall

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

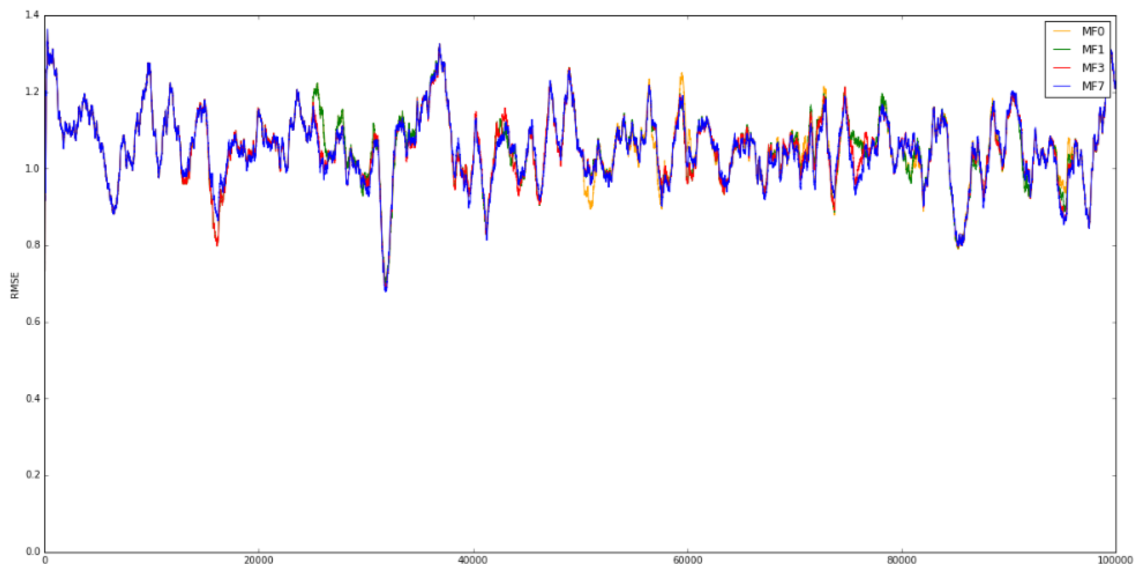

Fig. 13.: RMSE does not seem to be affected during the first 100k messages (MF with SGD)

The plots are essentially the same for the other models too.

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 5th strategy, that includes 7 updates, but only adds a user or item, if it has at least 5 ratings that belong to it.

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

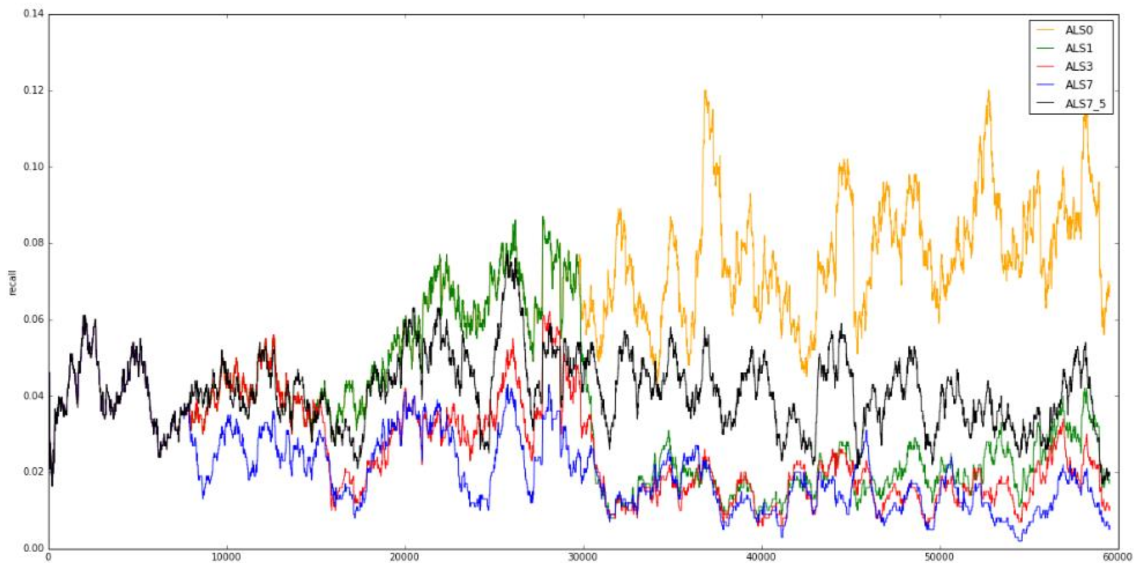

Fig. 14.: Setting a minimum rating limit for users/items can lead to better results

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.

8 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.

8.1 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.

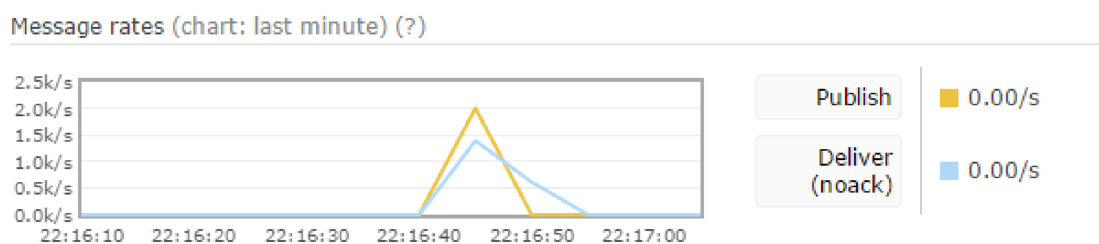

Fig. 15.: Messages are processed almost immediately

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 messages from the queue. Otherwise, even when an update is in progress, the usage decreases.

8.2 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 5) are not even noticeable.

9 Conclusions

The main issue of the essay, which is creating a model that uses the results of other models to give better recommendations seems to be a success. The RMSE of the combined model could stay below the minimum of the other models, and the recall could stay above the maximum. Since the combined model does not use any information on how the models work, it can be used with any model. In case the combined model is presented models that perform a lot better than the ones seen in this essay, it should still be able to improve their overall results.

It was also seen, that k-Nearest-Neighbors might not be the best choice for session-based systems. Producing all recommendations can be very slow, generally 1,000-10,000 times slower.

Observing the retraining schedules gave 2 surprising results: first, that more frequent updates don’t necessarily mean longer runtime. In fact, they may even speed things up. And second, that more personalized recommendations don’t necessarily mean better recommendations. This is one of the areas that should be further inspected in the future: how much information should the model have about a user by the time it starts giving personalized recommendations?

Finally, running multiple simulations showed that the recommender can perform well even under high stress. It can process new data even with a very high rate of requests. At the same time, giving recommendations is not an issue, since it is handled by a separate application, on a separate server, that only reads the recommendations from the database, that were put there by the recommender during the last update.

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

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

Users

Field name	*Description*
id	integer, serves as a unique identifier. In the case of the users, it *is* sequential
gender	its value is either ***F*** for female, or ***M*** for male
age	integer, indicating the age group of the user
occupation	integer, indicating the occupation of the user
zip code	integer, the user's zip code

Table 6.: The "users" dataset

Values	*Description*
1	younger, than 18 years
18	18-24 years
25	25-34 years
35	35-44 years
45	45-49 years
50	50-55 years
56	56 years, or older

Table 7.: User age groups

Values	*Description*
0	other" or not specified
1	"academic/educator"
2	"artist"
3	"clerical/admin"
4	"college/grad student"
5	"customer service"
6	"doctor/health care"
7	"executive/managerial"
8	"farmer"
9	"homemaker"
10	"K-12 student"
11	"lawyer"
12	"programmer"
13	"retired"
14	"sales/marketing"
15	"scientist"
16	"self-employed"
17	"technician/engineer"
18	"tradesman/craftsman"
19	"unemployed"
20	"writer"

Table 8.: User occupations

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