Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

Forecasting in Recommender Systems

Prognozowanie w Systemach Rekomendujących

Marcin Gruza

Praca inżynierska

Promotor : dr hab. Piotr Lipiński

Wrocław, Wrzesień 2019

Abstract

In some areas using recommender systems there exists time variability of products' attractiveness. In this work I present how to use forecasting to improve accuracy of algorithms in such systems. This approach can be used in any collaborative filtering algorithm with any forecasting model. I test my approach on MovieLens 20M dataset improving prediction results for few collaborative filtering algorithms.

Streszczenie

W wielu obszarach wykorzystujących systemy rekomendujące występuje czasowa zmienność atrakcyjności produktów. W tej pracy prezentuję wykorzystanie prognozowania do ulepszenia dokładności algorytmów w takich systemach rekomendujących. Podejście można stosować w każdym algorytmie zbiorowej filtracji wykorzystując dowolny model prognozowania. Swoje podejście waliduję na zbiorze MovieLens 20M, poprawiając wyniki predykcji dla kilku algorytmów zbiorowej filtracji.

Chapter 1

Introduction

Contextual information of user-item interactions in recommender systems can significantly improve recommender prediction accuracy. The examples of interaction's context are location, age of user or time, which seems to be the most important context that recommender system can use.

[3] suggests using additional time bin bias parameter for each item in standard matrix factorization algorithm. In that approach ratings of each product are grouped into time bins and each time bin has its own bias parameter $b_i(timebin)$. Authors use 50 time bins in a Netflix dataset which spans about 6 years of data. The prediction $R(u, i, timebin)$ for user u, item i and time bin $timebin$ is

$$R(u, i, timebin) = R(u, i) + b_i(timebin) = \mu + q^T p + b_u + b_i + b_i(timebin) \tag{1.1}$$

where μ is global mean, q, p are item and user feature vectors and b_i, b_u are item and user biases. The learning phase is the same as in standard matrix factorization algorithm. We can use stochastic gradient descent or similar to optimize all parameters. I've implemented this method to compare it to my approach.

The drawback of this approach is that it enforces using concrete matrix factorization algorithm to implement it. Moreover, this method can only model historic data and it doesn't forecast

time-dependent drifts in future. This can be problem when we have product with decreasing popularity as we don't have enough information to learn biases for the newest time bins.

This work focuses on using time-contextual information to forecast drifts in product attractiveness over time. I implement forecasting system recommender, which can use any standard forecasting model to improve any collaborative filtering algorithm. A free choice of forecasting model allows the model to forecast time-dependent drifts for the future and for periods with small popularity of product with very good accuracy. My method isn't limited to any algorithm and it can forecast item variability for any period of time.

Chapter 2

State of the Art

2.1 Recommender Systems

[5] Recommender systems are systems that try to predict interests of particular user and show him useful recommendations. These systems are very important in areas like online stores, social media, music or video streaming services and many other. They allow save a lot of users' time of searching items they will enjoy. Because of that system recommenders increase a chance of buying something or that the user will come back to the service later. This is why these systems are constantly being improved and new approaches are being created. Recommender systems can predict various types of user interactions with items like clicking, liking, rating or even buying a product.

There are two main approaches that recommender systems can use: content-based and collaborative filtering.

2.1.1 Content-Based Methods

Content-based methods use item features and properties to find similar items in database and recommend them to user. For example, if user likes movie "The Lord of the Rings" then system recommender can look for other movies within fantasy category or movies directed

by the same director. Recommender can also use some demography information about users - for example their home city, age, sex, etc. This method has a drawback - it usually has low variety of recommendations because of limited feature information about items. Also this method often requires manual providing information about item. This can be very painful in big and constantly growing systems. Another problem is that these methods don't consider item quality, as it usually can't be provided as item feature. This leads to recommending low quality similar items, which isn't always expected.

2.1.2 Collaborative Filtering

Collaborative Filtering is a way of predicting user preferences basing on preferences of other users. The main assumption of this method is that similar users have similar preferences. If user A has rated some items with ratings very similar to user B, then they probably have similar taste and will like the same items in the future. The main advantage of this method is that it doesn't require any feature information about an item. It only uses item and user identification tokens. The drawback of this method is called the cold start problem. This problem appears when there is new item or user in the database. If this is item, we don't have any votes or interactions made with it so we can't find similar items. The same problem arises when we have new user in the system. Another problem is sparsity of the data. In a system where we have millions of items, users usually interact only with few of them. Because of this, many items have very small amount of votes and we can't make any useful predictions about them. Despite some problems, collaborative filtering is main method used in recommender systems and in this work I will focus on improving algorithms implementing this method.

There are many collaborative filtering algorithms falling into two categories: **model-based** and **memory-based**.

Memory based collaborative filtering algorithms use vectors of user-item interactions directly to make new predictions. An example of this kind of algorithm is K nearest neighbours algorithm. This algorithm simply finds most K similar users/items in system and averages their ratings to get new prediction. This approach has a problem of low scalability. Data in a typical

system is very sparse and there are a lot of items, which leads to big memory usage of these algorithms.

Model based collaborative filtering algorithms create a model based on ratings data in the system. It can use many machine learning algorithms and models to do that. Usually these algorithms do some dimensional reduction which is very beneficial as it allows to save memory. Singular value decomposition method is one of the most popular model based algorithm and it usually achieves the best accuracy.

2.1.3 Problem Definition for Collaborative Filtering

As described in [4], there are few options of defining user preference in recommender system. For example, we can define it as a chance that user will visit a page of some product, like it or buy it. Another possibility is to define it as a rating that user will give to the item. For formal definition of the problem, we will denote W as a set of possible interaction values between user and item. It can be either set of possible rating values ($W = 1, ..., 5$) or kind of interaction ($W = \{dislike, like\}$, $W = \{notvisited, visited\}$). For simplicity, we will assign a real number to each kind of interaction value, where more positive interaction will have bigger number assigned. Let $\mathcal{U}$ denote a set of users, $\mathcal{I}$ a set of items, and $\mathcal{R}$ a set of user interactions with items in a system, where $r_{ui} \in \mathcal{R}$ denotes an interaction of user u with item i. The typical problem recommender system tries to solve is called top-N recommendations problem. This means finding N items that the user u will like the most. In a formal way, we want to find set A of N items, such that:

$$A = \underset{\{a_1,a_2,...,a_N\} \subseteq I}{\arg\max} \sum_{k=1}^{N} R_{ua_k} \tag{2.1}$$

2.1.4 System Recommender Evaluation

First step to measure accuracy is to split dataset into two parts: training and test. The training set $R_{training}$ is what we will use to make some predictions. Then we will compare these

predictions with true ratings in test set R_{test} with some evaluation metrics. Two most popular accuracy metrics to evaluate recommender system are RMSE (Root Mean Squared Error) and MAE (Mean Absolute Error). Let $\hat{r}_{ui}$ denote recommender system estimation of interaction that user u done with item i. Then RMSE is defined as

$$RMSE = \sqrt{\frac{\sum_{r_{ui} \in R_{test}} (r_{ui} - \hat{r}_{ui})^2}{|R_{test}|}} \tag{2.2}$$

and MAE is defined as

$$MAE = \sum_{r_{ui} \in R_{test}} \frac{|r_{ui} - \hat{r}_{ui}|}{|R_{test}|} \tag{2.3}$$

The difference between these two metrics is that RMSE penlizes system recommenders which make bigger errors. For example, if the reference ratings are equal (2,2,2) and recommender system predicts (4,2,2) then RMSE = 4 and MAE = 2. But if the prediction is (3,3,2), then RMSE = 2 and MAE = 2.

Another kind of system recommender evaluation metrics are ranking-based measure metrics. ths kind of metrics measures how much order does the system recommender keep between ratings. Usually we don't want to know exact rating that user will give to item. Instead, we want to know which of them will be most liked by him. This is why order between ratings is very important. Two of such metrics are: **Normalized Distance-based Performance Measure (NDPM)**[1] and **Fraction of Concordant Pairs (FCP)** [2].

2.1.5 KNN

The most popualar model-based collaborative filtering algorithm. This Neighborhood-based model can use item-item or user-user approach. In the first case, when algorithm makes prediction for user u and item i, it looks for k most similar items to item i. Let $sim(i_j, i_k)$ denote similarity between items i_j and i_k, $I_{sim(i),k}$ denote k most similar items to item i. Then our

prediction is weighted average of ratings made by user u to similar items:

$$\hat{r}_{ui} = \sum_{i_j \in I_{sim(i),k}} \frac{sim(i, i_j)}{\sum_{i_j \in I_{sim(i)}} |sim(i, i_j)|} r_{ui_j} \tag{2.4}$$

The similarity is divided by sum of all similarities to force that all weights sum to 1.0.

Similarly, in user-user approach, when making prediction for user u and item i, we find k most similar users to user u. Let $sim(u_j, u_k)$ denote similarity between users u_j and u_k, $U_{sim(u),k}$ denote k most similar users to user u. Then our prediction is weighted average of ratings made by most similar users to item i:

$$\hat{r}_{ui} = \sum_{u_j \in U_{sim(u),k}} \frac{sim(u, u_j)}{\sum_{u_j \in U_{sim(u)}} |sim(u, u_j)|} r_{u_j i} \tag{2.5}$$

This algorithm needs a definition of a similarity function. One such function is cosine similarity. The cosine similarity between vectors w and v is defined as:

$$cos(w, v) = \frac{w^T v}{||w||||v||} \tag{2.6}$$

2.1.6 Baseline

The baseline predictor is simple model-based collaborative filtering algorithm which models only biases of both users and items. Each item i has its own bias parameter b_i and each user has its own bias parameter b_u. Prediction for rating made by user u on item i is equal

$$r_{ui} = \mu + b_u + b_i \tag{2.7}$$

where μ is global mean of train dataset. Bias values can be found with simple optimization algorithms like stochastic gradient descent or alternating least squares, by optimizing error

$$Err = \sum_{r_{ui} \in R} (r_{ui} - \mu - b_u - b_i) \tag{2.8}$$

where the sum goes through all ratings in a train set. This model is very simple and it doesn't take into account user-item relationship. Because of this, it doesn't make sense to use it in real recommender system, but we can check how important are item and user biases in a system.

2.1.7 SVD

Let d be feature vector length, n be number of users and m number of items in a system. Main idea of matrix factorization collaborative filtering algorithms is to decompose matrix $R \in \mathbb{R}^{nxm}$ of ratings to two matrices: user matrix $Q \in \mathbb{R}^{nxd}$ and item matrix $P \in \mathbb{R}^{mxd}$, such that

$$QP^T \approx R \tag{2.9}$$

The SVD is an example of simple matrix factorization collaborative filtering algorithm which usually turns out to be very accurate model. In this model each user and item are mapped to a vector space. Let $q_u \in \mathbb{R}^{1xd}$ denote vector representing user u and $p_i \in \mathbb{R}^{1xd}$ vector representing item i. Dimensions of these vectors may represent some features, like category. Then, we treat product $q_u {p_i}^T$ of these vectors as their relationship, which may be positive or negative. Additionally, we keep bias parameters for each item and user as in baseline algorithm. The prediction of rating made by user u to item i is

$$\hat{r}_{ui} = \mu + q_u {p_i}^T + b_i + b_u \tag{2.10}$$

Despite its name, the algorithm doesn't use standard SVD method of matrix factorization, as SVD cannot factor matrix with missing values. Instead, the algorithm updates weights of Q, P and biases while iterating all ratings in train part of set. The update can be done either by

alternating least squares or stochastic gradient descent, by optimizing error

$$Err = \sum_{r_{ui} \in R} (r_{ui} - \mu - q_u {p_i}^T - b_u - b_i) \tag{2.11}$$

2.1.8 Deep Learning Models

Contemporary recommender systems often use deep neural network as collaborative filtering matrix factorization algorithm. To do that, they use embedding vectors to represent user and item features. They allow to create more flexible and bigger models. They are also more scalable as they allow GPUs computation.

These are most popular collaborative filtering methods used in contemporary recommender systems and their benchmarks. Because matrix factorization methods are more scalable and they have lower memory usage, in experiments of this work I use only these algorithms.

Chapter 3

Problem Definition

Most of recommender systems use item bias to predict exact item rating. This bias reflects attractiveness of an item compared to other items. However, as the attractiveness of item may change over time, item bias in the model should reflect these changes.

For example, if we predict popularity of Home Alone movie, we expect the bias of this item to be much bigger at Christmas than in the summer. Another example is popularity of IPhone 5 phone, which was very popular and liked by users when it appeared in the shops, but we don't want to recommend it too often now, as people prefer newer phone models. Also, global preferences of users change over time and our model should take it into account.

Standard collaborative filtering algorithm ignores time changes and it usually averages the item bias over time range of all ratings. Prediction of rating $\hat{r}_{ui}$ for user u and item i is always the same, no matter when we make this prediction. One possibility of updating the model to reflect time changes is to ignore old samples in the system, but this approach can decrease model accuracy because we loose some information. Instead, we want to extend our model to take into account exact time of given rating and improve accuracy basing on it. The problem is how to incorporate time-changing factor into model prediction to improve model accuracy.

Chapter 4

Using Time-Context in Recommender Systems

4.1 Idea

In this work I suggest using a forecasting model to forecast item average rating variability over time and use this model to improve our recommender system model. First step of this approach is to create forecasting model of mean item ratings. For each item i we create item's average rating model $\mu_i(t)$, where t is a timestamp. For items that may not have clear and useful time variability we may use simple single item global mean model.

Second step is to create new representation of ratings in a training set. Each rating will be transformed into a deviation from item average rating forecasted for time of sample timestamp with model created in first step. This deviation can be interpreted as how much the user like this item relative to the item's typical attractiveness. The deviation for an item rating is computed as:

$$dev(r_{ui}(t)) = r_{ui} - \mu_i(t) \tag{4.1}$$

where $r_{ui}(t)$ is rating made by user u to item i in time point t and $\mu_i(t)$ is forecasted mean item rating in time t.

These deviations can now be treated as raw ratings and used as standard input to any known collaborative filtering algorithm. The algorithm predictions will also be deviations so we have to add this predicted deviation to forecasted item average for the time we want to do prediction. Our prediction of item rating becomes

$$\hat{r}_{ui}(t) = \hat{dev}(r_{ui}(t)) + \hat{\mu}_i(t) \tag{4.2}$$

where $\hat{dev}(r_{ui}(t))$ is forecasted deviation from item i average rating in time t and $\hat{\mu}_i(t)$ is forecasted mean item rating in time t.

This approach allows to model time variability for any period of time, which is helpful for items with decreasing popularity because we may have very small amount or no ratings of given item in some time periods. The time-aware recommender system model mentioned earlier models time variability only for time periods with some samples within it, which may be insufficient. Another advantage of this method is that we can forecast average item rating only for chosen items, for which we expect strong, clear and easy to model time variability.

4.2 Forecasting Model

The forecasting model should be chosen after collected data is analyzed and all significant data patterns are found. For more time dependent systems with strong trends and clear seasonality we can use sophisticated forecasting models like ARIMA. For systems with soft trend and no seasonality we can use simple forecasting models like last sample model.

4.3 Collaborative Filtering Algorithm

The idea is very universal and probably all collaborative filtering algorithms can use it. However, experiments show that not all algorithms can be improved with this approach. Therefore we should always compare performance of our algorithm with and without forecasting method.

Chapter 5

Validation of the Approach

5.1 Data

In this section I'll describe dataset used to validate my approach. I'll check some basic properties and a time dependence of data. This is important step before choosing forecasting model in our method.

5.1.1 MovieLens20M

MovieLens20M is a movie ratings dataset, released on 4/2015, commonly used as recommender system benchmark. It consists of 20 millions ratings made by 138000 users to 27000 movies. Each rating consists of user id, movie id, rating value (0.5-5.0) and a timestamp. The data was gathered between January 09, 1995 and March 31, 2015. Each user in the dataset rated at least 20 movies and each movie has at least one rating. The data contains some information about movies: title, year of release, tags and categories. There is no additional information about users.

Movie Id	Title	Genres	#ratings	Avg. rating
296	Pulp Fiction (1994)	Comedy, Crime, Drama, Thriller	67310	4.17
356	Forrest Gump (1994)	Comedy, Drama, Romance, War	66172	4.02
318	Shawshank Redemption, The (1994)	Crime, Drama	63366	4.44
593	Silence of the Lambs, The (1991)	Crime, Horror, Thriller	63299	4.17
480	Jurassic Park (1993)	Action, Adventure, Sci-Fi, Thriller	59715	3.66

Table 5.1: 5 most popular movies in the MovieLens20M dataset with their rating counts and average rating.

5.2 MovieLens20M Dataset Analysis

In this section, I'll analyze MovieLens20M. First I'll analyze ratings distribution and then I'll focus on time analysis of the dataset.

5.2.1 Ratings Distribution

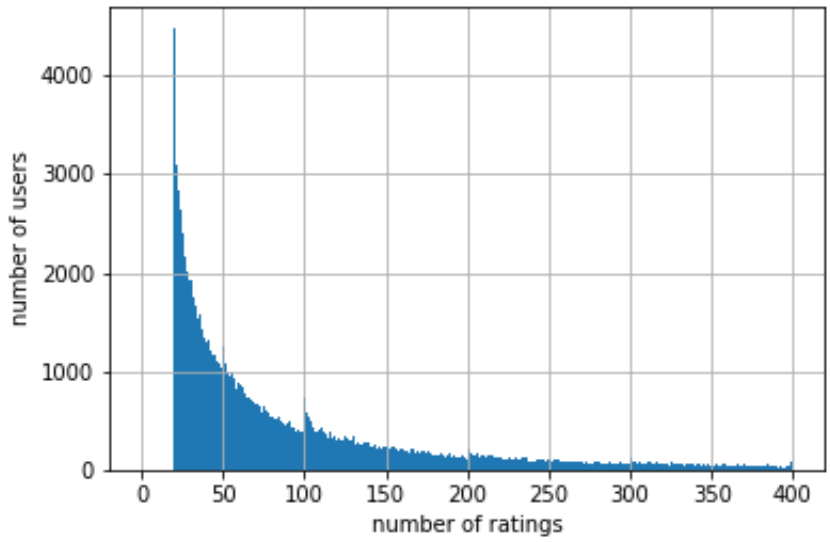

Figure 5.1: Number of ratings made by one user.

A movie in the dataset has 748 ratings on average, but the median number of ratings per movie is equal 18. This means that there is a lot of movies with very small amount of ratings which can lead to problems with making useful forecasting for them, therefore we should probably filter them out and forecast movie ratings only for more popular movies. Average number of ratings done by a user in a dataset is 144, and the median is 68. Global mean of ratings is equal 3.5255 and standard deviation is 1.05.

5.2.2 Time Analysis

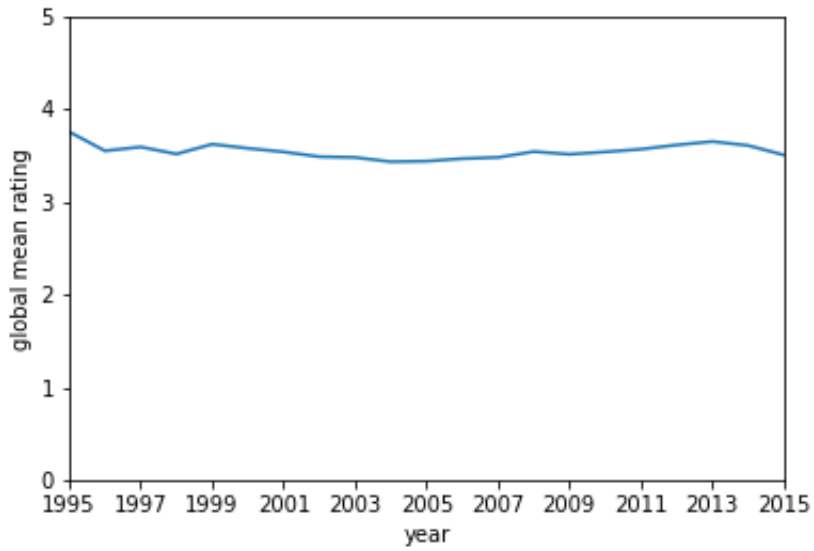

Figure 5.2: Global mean of ratings over time.

First, I'll check global time variability of the dataset. Whole dataset ranges over 7385 days. Figure 5.2 shows that there is no significant global mean of ratings change over time. This shows that people don't change their general attitude towards movies.

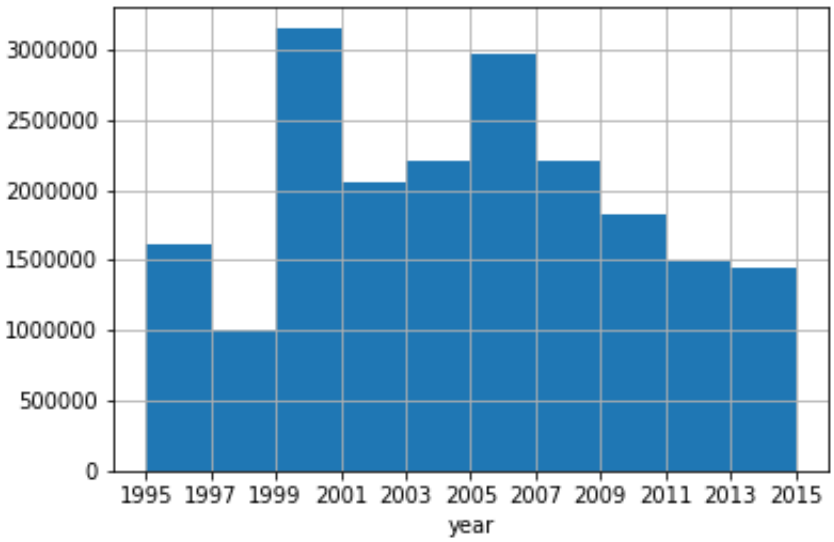

Figure 5.3: Number of ratings done by year.

The number of new ratings per year is quite uniformly distributed, except the year 1995 which has only 4 ratings. Figure 5.4 shows that the last few years has the biggest number of new

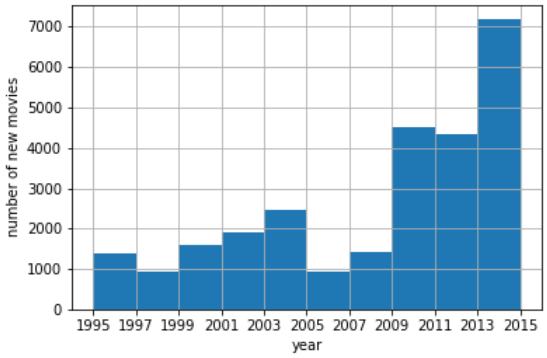

Figure 5.4: Number of new movies per year.

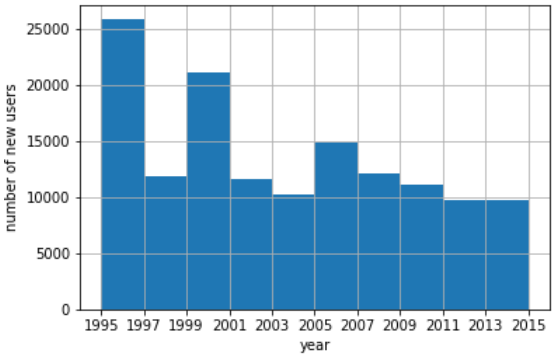

Figure 5.5: Number of new users per year.

movies added to dataset. Number of new users per year is uniformly distributed, with some peaks on first few years of data.

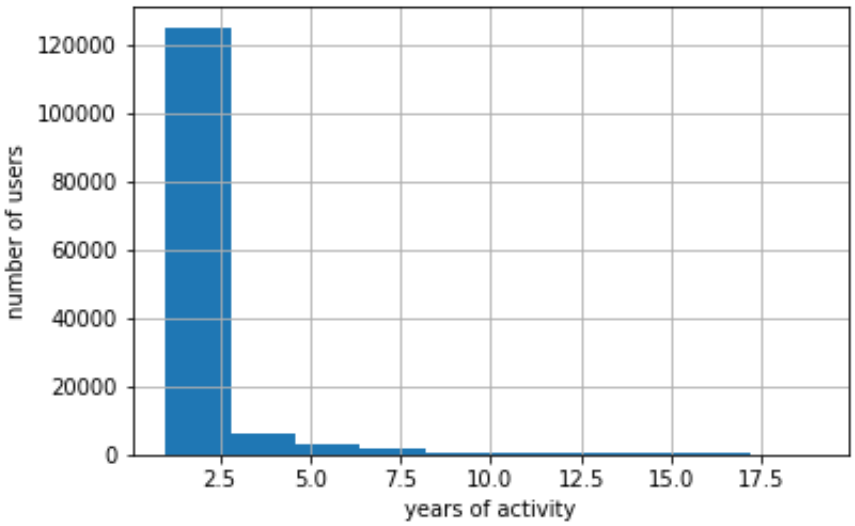

Figure 5.6: Number of ratings done by year.

Figure 5.6 shows that most of users are active for one or two years, where active time is measured as number of years between their first and last rating made in a system. Average number of years of user's activity is equal 1.55.

Now, we'll check more local time variability in a dataset. Two most interesting things are average rating over time and seasonality of particular movie.

To analyze movie ratings over time, we have to aggregate ratings first, because time series of raw ratings are too chaotic. Therefore, movie ratings are aggregated into 30-day time bins, and average of ratings in these time bins are being used to analyze the data. Figure 5.8 shows mean

rating of movies *Gran Torino* and *Into the Wild* over time.

Figure 5.7: Average rating of of ratings aggregated into 30 days time bins for movies *Gran Torino* and *Into the Wild*. Histogram on the plots represents number of ratings for each time bin.

First, let's analyze some movies seasonality. Table 5.2 shows average monthly ratings for two of most popular movies in dataset. As we can see, the differences between the mean ratings for months are very small. This suggests no significant monthly seasonality of movie ratings in the dataset. Furthermore, Figure 5.8 of mean movie ratings over time with 30 day time bins show no clear seasonality patterns in them. The fact that movie ratings have no seasonality seems reasonable. Typically no matter when we see a movie, we have similar feelings about it. The thing that varies and may have some seasonal pattern is movie popularity. For example

Christmas movies are much more popular in winter than in other seasons. However, predicting popularity is a different problem than predicting exact user ratings.

The trend of movie rating mean over time is noticeable in time bin aggregated movie ratings. To make them more clear and simple, I used EWMA (exponentially weighted moving average) to smooth time series of average movie ratings. Figure 5.7 shows smoothed time series of average movie ratings for two popular movies. The trend is not very strong and clear, but it's noticeable.

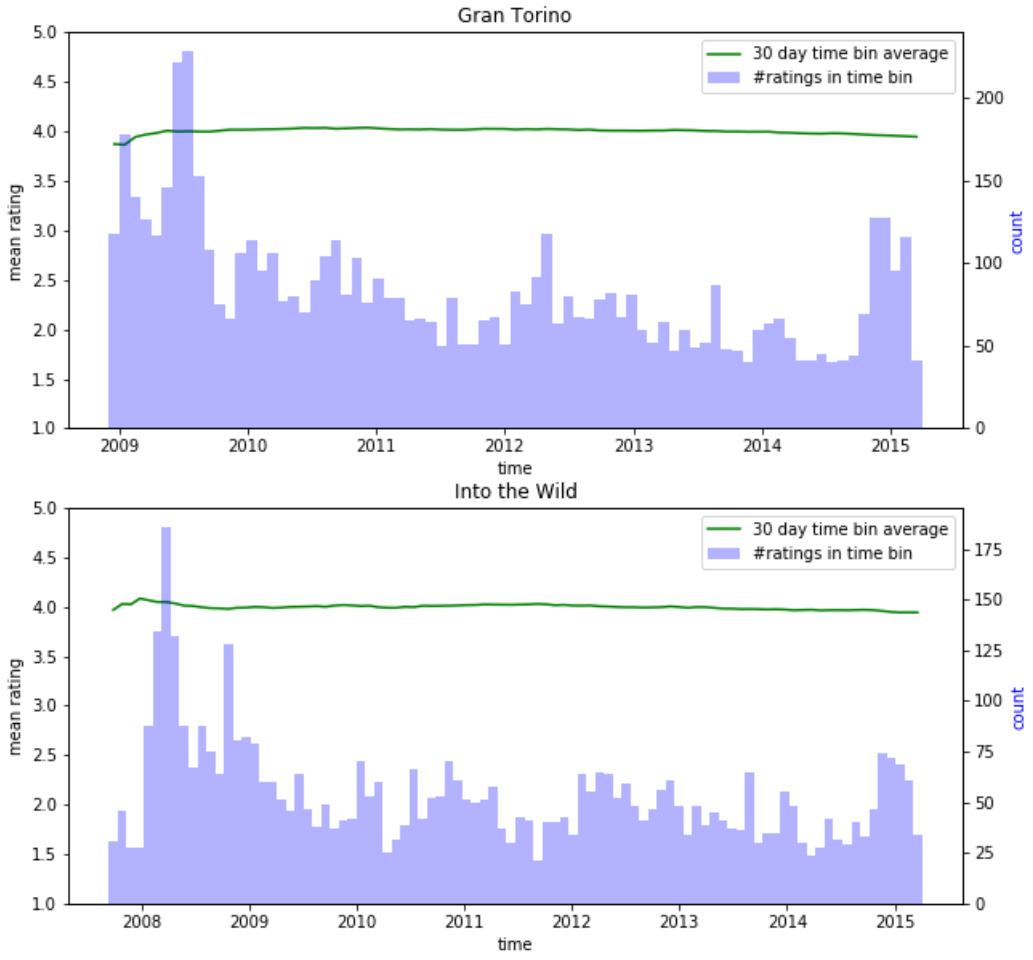

Figure 5.8: Average rating of of ratings aggregated into 30 days time bins for movies *Gran Torino* and *Into the Wild* smoothed with EWMA ($halflie = 50$). Histogram on the plots represents number of ratings for each time bin.

Time analysis of the MovieLens data indicates that the average movie rating over time is simple random walk model. This model can't be reasonably predicted by any advanced time models

month	Shawshank Redemption, The	Forrest Gump
January	4.43	4.02
February	4.42	4.01
March	4.40	3.92
April	4.48	4.00
May	4.43	4.02
June	4.46	4.06
July	4.47	4.07
August	4.46	4.05
September	4.44	4.03
October	4.42	3.99
November	4.45	4.05
December	4.45	4.03

Table 5.2: Mean monthly rating of two popular movies. Mean for each month is calculated from all ratings made in this month since the movie appeared in dataset.

and because of that I've decided to use simple last sample model to forecast movie mean ratings for given time. Another reason why we don't need any sophisticated forecasting model is that, we usually have very fresh data about given item in system, so we don't need to forecast anything for a long time in the future.

5.3 Experiments

To evaluate approach presented in this work, I test it on MovieLens20M dataset using few popular collaborative algorithms: Baseline, SVD, Probabilistic Matrix Factorization and Neural Network Matrix Factorization algorithm implemented by me, described in appendix.

5.3.1 Environment

All the experiments was performed on Google Colaboratory, free cloud Jupyter notebook environment with about 25 GB RAM. The environment was set to using Python 3 and GPU acceleration to make neural network algorithm computations faster.

5.3.2 Software

I used Surprise SciKit (version 1.0.6) and its basic algorithm implementations. Surprise also contains accuracy metric implementations: RMSE, MAE, FCP. The NDPM metric implementation is included in my source code. Other used libraries are numpy, pandas and matplotlib to visualize some data. The neural network algorithm was implemented using PyTorch library (version 1.2).

5.3.3 Dataset Splitting Methods

In experiments I use three methods of splitting dataset into train and test parts. This simulates various system ratings distribution in time which may have impact on forecasting and therefore on results.

1. Cross-validation on randomly shuffled data. This is the simplest and most popular way of splitting data to test a model. This method splits data randomly into k equal parts and uses one of them as test part while remaining parts are used as train part. This procedure is repeated k times, with different part used as test data each time.

2. Splitting data by some time point t. All ratings made before t are used as train part, while all newer ratings are used as test part. This method simulates case where we have some historical data and we want to use it in new recommender system.

3. Popularity decay split. In this method, we group ratings by movie and throw away 80 % of ratings done later than 3/10 of given movie lifetime. For example, if movie m oldest rating was made in 2000, and the newest in 2010, then we throw away 80 % randomly chosen ratings made in year range 2003-2010. This simulates datasets with high and fast decrease of products' popularity. Filtered data is splitted into training and test parts randomly.

5.3.4 Forecasting

Because movielens dataset contains many movies with small number of ratings, the forecasting of movie means was made only for 4000 most popular movies in dataset. Mean over time for these movies were modeled in following way:

1. Ratings of given movie was grouped by consecutive 10 day time bins and mean of ratings in each time bin was calculated. These means is now treated as time series of this movie average rating.

2. Next, we throw away all time bins with less than 5 ratings inside, as they may be outliers.

3. Finally, we smooth time series with EWMA. Halflife parameter of EWMA is set to 50.

For the rest of movies we assume that their mean is static over time and equals total mean of given movie in train set.

5.3.5 Results

Results are summarized in tables 5.4, 5.3, 5.5. As we can see, using forecasting improves results for standard cross validation dataset split method in all used recommender system models. Data in this split method is most uniformly distributed and therefore forecasting predictions are reliable.

Results for second and third data split method are also mostly improved. Neural networks benefits most from the forecasting method. Magnitude of improvement is quite low due to very low time-variability of movie means in MovieLens dataset. Still, this improvement proves that there is some time-dependency in the data and the approach models it.

Results for time-awareness system recommender from handbook doesn't give significant improvements as presented in Table 5.6.

The results show that the approach I presented in this work improves overall accuracy of recommender systems. The low magnitude of improvement is caused by low time variability of movie rating means over time. Despite the simple model and unclear changes over time, the method still improves most of models accuracy.

The only algorithm which gives significant better results without forcecasting method is PMF on popularity decay splitted data. This is probably caused by no biases in the PMF model, and distribution of users in this data split method - almost all users from test dataset are known in trainset.

Results on popularity decay split method give the least improvements of accuracy. This is probably caused by less accurate forecasting of mean when we have less samples in the future. Movie means are hard to forecast because of very soft trend and no clear time patterns.

	RMSE (less is better)		MAE (less is better)		NDPM (less is better)		FCP (more is better)	
Forecasting	no	yes	no	yes	no	yes	no	yes
Baseline	0.859	0.856	0.660	0.658	0.316	0.312	0.711	0.712
SVD	0.811	0.802	0.624	0.613	0.296	0.286	0.744	0.748
PMF	0.837	0.828	0.648	0.638	0.298	0.289	0.735	0.737
Neural network based	0.810	0.809	0.619	0.618	0.286	0.284	0.7448	0.745

Table 5.3: Results for cross validation split method. Evaluation metrics are defined in 2.1.4.

	RMSE (less is better)		MAE (less is better)		NDPM (less is better)		FCP (more is better)	
Forecasting	no	yes	no	yes	no	yes	no	yes
Baseline	0.942	0.936	0.713	0.715	0.316	0.312	0.711	0.712
SVD	0.931	0.932	0.714	0.711	0.352	0.353	0.627	0.628
PMF	1.010	0.943	0.778	0.719	0.484	0.355	0.410	0.623
Neural network based	0.955	0.933	0.723	0.713	0.372	0.353	0.610	0.627

Table 5.4: Results for split by time point method. Evaluation metrics are defined in 2.1.4.

	RMSE (less is better)		MAE (less is better)		NDPM (less is better)		FCP (more is better)	
Forecasting	no	yes	no	yes	no	yes	no	yes
Baseline	0.856	0.851	0.643	0.642	0.328	0.323	0.662	0.665
SVD	0.8178	0.8179	0.6139	0.6143	0.3116	0.3094	0.6818	0.6823
PMF	0.855	0.879	0.634	0.650	0.308	0.310	0.681	0.673
Neural network based	0.818	0.816	0.615	0.613	0.306	0.304	0.686	0.687

Table 5.5: Results for popularity decay split method. Evaluation metrics are defined in 2.1.4.

	RMSE (less is better)		MAE (less is better)		NDPM (less is better)		FCP (more is better)	
Forecasting	no	yes	no	yes	no	yes	no	yes
Neural network based	0.80996	0.80864	0.61969	0.61913	0.28616	0.28375	0.74430	0.74339

Table 5.6: Results for cross validation of time-awareness matrix factorization method. Evaluation metrics are defined in 2.1.4.

Chapter 6

Conclusions

6.1 Summary of Thesis Achievements

In this work I show how to use forecasting to improve recommender system accuracy. The method improve accuracy of systems where time dependency of data is present. Results show that even if time variability isn't very significant, it can still improve recommendation accuracy. It improves RMSE prediction error on MovieLens 20M dataset from 0.811 to 0.802 for SVD model and from 0.810 to 0.809 for Neural Network model. It's difficult to compare these results to other approaches because RMSE prediction errors values for MovieLens range from 0.77 to 0.81 ([6]), and they depend on training-test dataset ratio, data subset used to train a model and data preprocessing. But another item time-context collaborative filtering technique presented in [3] improves RMSE error from 0.9799 to 0.9771 for MovieLens 1M (0.80996 to 0.80864 for MovieLens 20M in my implementation), which is worse result than the one presented here for SVD model and comparable to result for Neural Network model.

6.2 Applications

This method can be used in any system recommender that logs time information about ratings done in history. The method is very efficient, as it can cache forecasting model for particular

products. Therefore this method seems to have some practical applications. Systems predicting a chance of user interaction with product will benefit a lot from this approach, as popularity of product is very time dependent and it has big impact on the possible interaction with given item.

Bibliography

[1] Yao, Y.Y. (1995). *Measuring retrieval effectiveness based on user preference of documents.*. J. Amer. Soc. Inf. Sys 46(2), 133–145

[2] Koren, Yehuda & Sill, Joseph. (2013). *Collaborative Filtering on Ordinal User Feedback* 3022-3026.

[3] Koren, Yehuda & Bell, Robert. (2015). *Advances in Collaborative Filtering.* 10.1007/978-1-4899-7637-6_3.

[4] Desrosiers C., Karypis G. (2011). *A Comprehensive Survey of Neighborhood-based Recommendation Methods.* In: Ricci F., Rokach L., Shapira B., Kantor P. (eds) Recommender Systems Handbook. Springer, Boston, MA

[5] Ricci, Francesco & Rokach, Lior & Shapira, Bracha. (2010). *Recommender Systems Handbook.* 10.1007/978-0-387-85820-3_1

[6] Florian Strub, Romaric Gaudel, Jérémie Mary. *Hybrid Recommender System based on Autoencoders.* the 1st Workshop on Deep Learning for Recommender Systems, Sep 2016, Boston, United States. pp.11 - 16, ff10.1145/2988450.2988456ff. ffhal-01336912v2f

[7] Collaborative Filtering, Wikipedia
`https://en.wikipedia.org/wiki/Collaborative_filtering`

[8] Surprise - scikit for recommender systems.
`http://surpriselib.com/`

[9] Pytorch - neural networks library.

`https://pytorch.org/`

[10] Colaboratory - Jupyter notebook environment.

`https://colab.research.google.com/`

Appendices

Appendix A

Neural Network Colaborative Filtering Algorithm

As a part of my thesis, to test my approach faster and in more flexible way, I implemented neural network matrix factorization algorithm. The algorithm is inspired by Word2Vec model.

A.1 Idea

Each user in the algorithm is represented with vector $q_u \in \mathbb{R}^{1xd}$ and bias $b_u \in \mathbb{R}$. Each item is represented with $p_i \in \mathbb{R}^{1xd}$ and bias $b_i \in \mathbb{R}$. d is feature vector length given as algorithm's parameter. The algorithm has an option to add time bin biases to items. With this option enabled, each item has additional vector of biases $b_{i,timebin} \in \mathbb{R}^{1xtimebins\_num}$, where number of time bins is given as algorithm's parameter. The rating prediction for user u and item i is computed in following steps:

1. Sum vector of user and item vector is calculated: $q_u + p_i$
2. The vector is applied to linear neural network layers specified by algorithm user.
3. After all layers applied, vector is projected into one dimension and biases are added to prediction.

A.2 Implementation

I implemented the algorithm with PyTorch library and adapted it to work with Surprise library, which means it can be trained and tested with Surprise in the same way as standard Surprise library algorithms.

Appendix B

Example of Movie Mean Changes Over Time

Here I present changing movie average over time for some popular movies in MovieLens dataset. Smoothed time series show soft but visible trends.

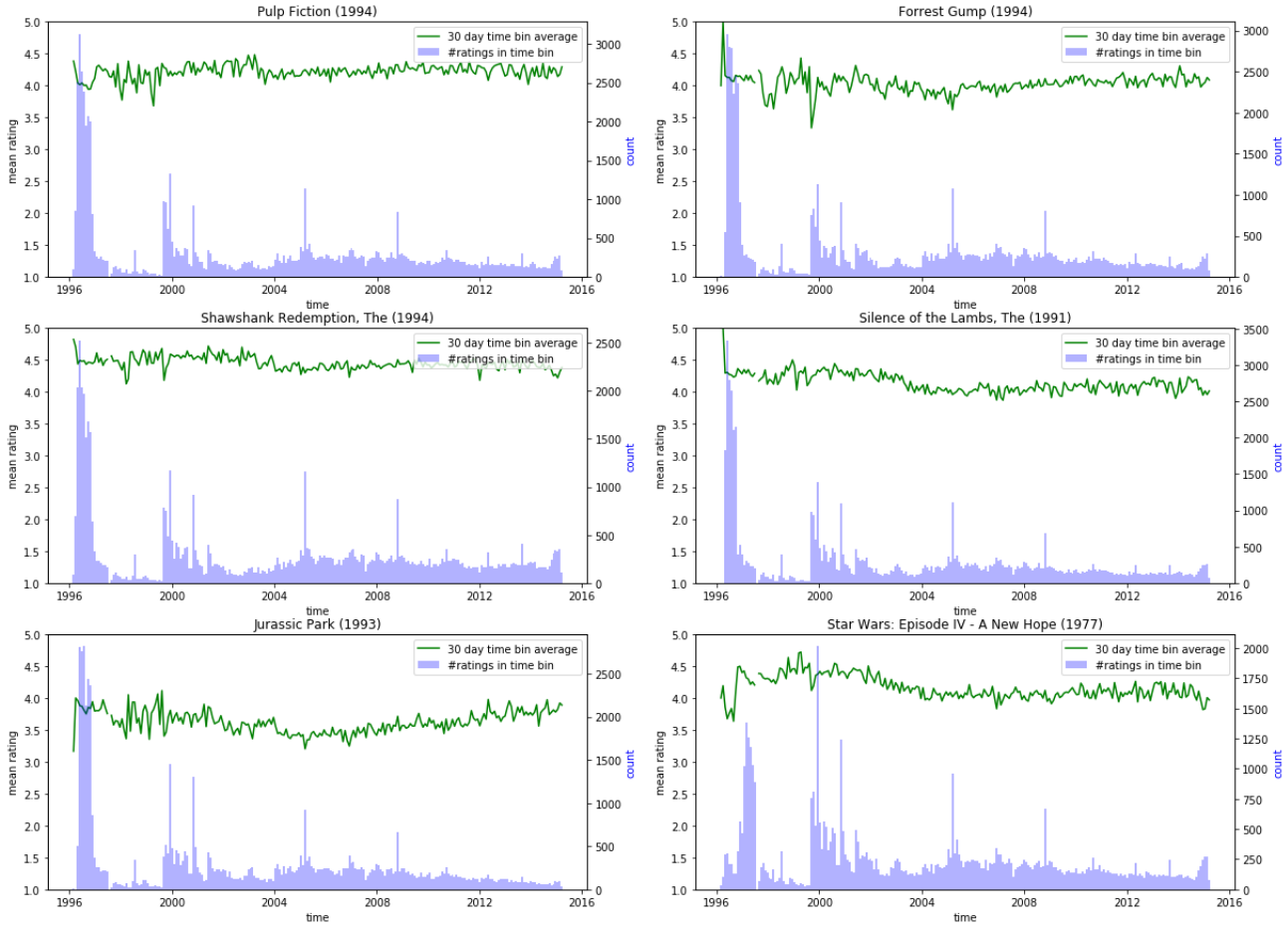

Figure B.1: Average rating of of ratings aggregated into 30 days time bins for 6 most popular movies in MovieLens. Histogram on the plots represents number of ratings for each time bin.

.

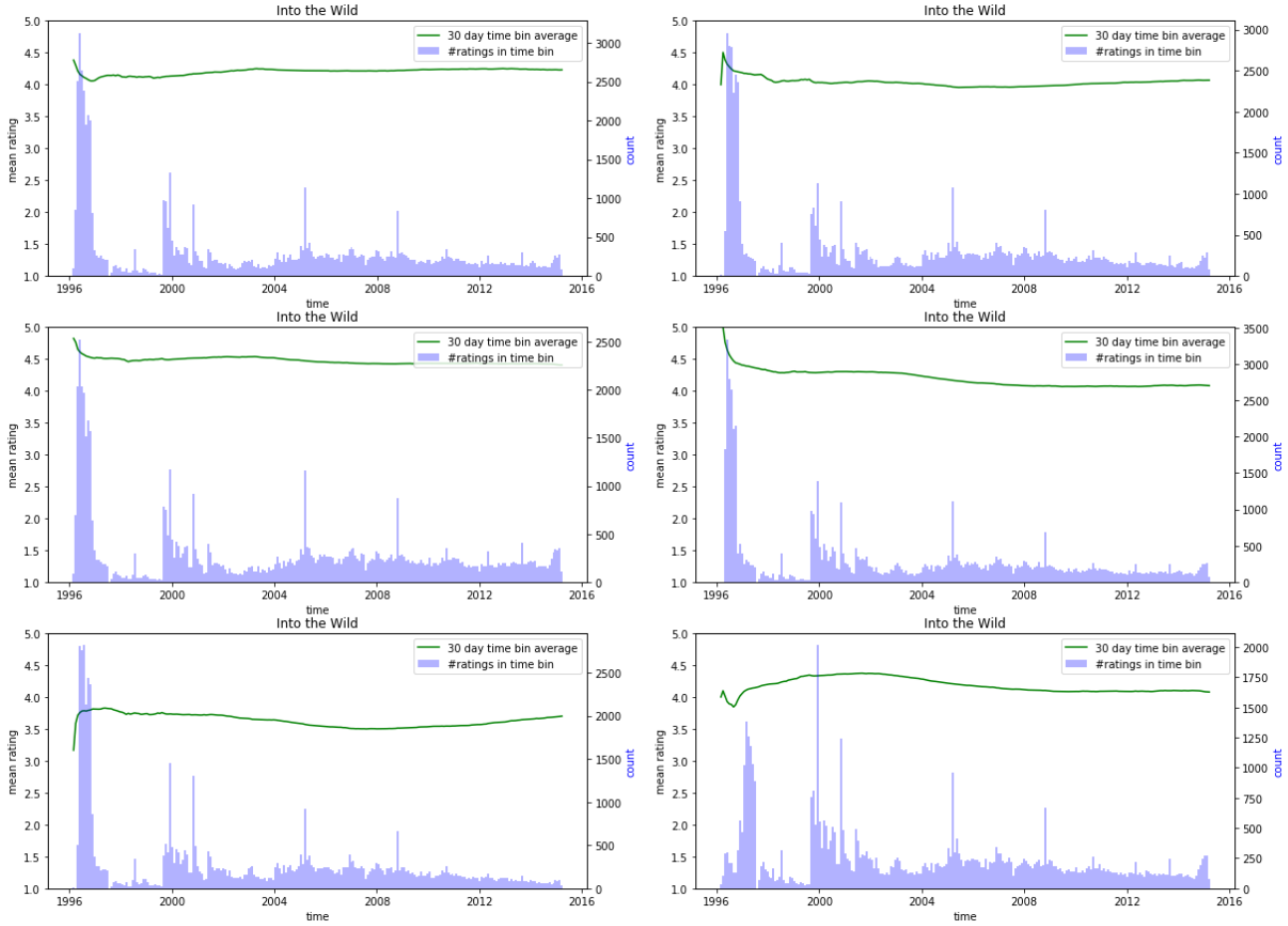

Figure B.2: Average rating of ratings aggregated into 30 days time bins and smoothed with EWMA ($halflife = 50$) for 6 most popular movies in MovieLens. Histogram on plots represents number of ratings for each time bin.

.

Appendix C

Instruction to Perform Simulations

C.1 Reproducing Results

All of experiments can be reproduced using notebook ForecastingInRecommenderSystems.ipynb. To execute scripts, user needs to install Surprise and PyTorch libraries.

C.2 Performing Own Experiments

Implementation of forecasting system recommender is in file source/recommender.py, class MeanDeviationsRecommender. To instiate it, we need base recommender model, forecaster and timestamps dictionary. Base recommender model has to implement standard surprise interface of recommender system algorithm : fit(trainset) and predict(u, i) methods.

The forecaster is an instance of RatingsForecaster class from the same module. This class models average movie rating over time. To instantiate it, we need forecasting model, for example LastStample model from source/forecaster.py module. Then we need to invoke compute_devs() method on forecaster object, to calculate all mean deviations for movies. Timestamps dictionary of the forecasting recommender is simple python dictionary, which maps tuples (user_id, item_id) to timestamp.

Here is example of code to perform own experiments:

```
import surprise
import numpy as np
import source.recommender as rcm
import source.datasets as ds
import source.utils as ut
import source.forecast as ft

#split data into train and test pandas dataframes
train_df, test_df = movielens20m.ratings[:10000000], movielens20m.ratings[100

#create surprise representations of data
train_dataset = ds.to_surprise_trainset(train_df)
test_dataset = ds.to_surprise_testset(test_df)

# get dataframe of 2000 most popular movies in trainset
most_popular_movies = ut.get_popular_movies_ids(train_df, movielens20m, 2000)

# create forecaster using LastSample model
forecaster = rcm.RatingsForecaster(ft.LastSample)

# compute deviations for given movie Ids
forecaster.compute_devs(train_df, most_popular_movies.movieId.tolist())

# compute all means for test samples and cache them. This works faster
# than forecasting mean each time we make a prediction.
forecaster.precompute_means(test_df)

# create test timestamps dictionary
```

```
test_timestamps_dict = ut.get_timestamps_dict(test_df)

# create basic surprise recommender system algorithm
algo = surprise.prediction_algorithms.baseline_only.BaselineOnly()

# create forecasting recommender system algorithm
f_algo = rcm.MeanDeviationsRecommender(algo, forecaster, test_timestamps_dict

# fit and test forecasting algorithm
f_algo.fit(train_dataset)
predictions = f_algo.test(test_dataset)


# evaluate predictions
surprise.accuracy.rmse(predictions)
surprise.accuracy.mae(predictions)
```