Recommender System Algorithms for Implicit Job Dataset

GERLANDO SAVIO SCIBETTA and MATTEO POZZI, Politecnico di Milano

This paper is a short summary of our efforts for this year's Recommender Systems competition. The result of our experiment is a Recommender System which provides better Recommendations than any of the single method and techniques. The algorithm were evaluated on both a private and a public test-set using the MAP@5 metric. To obtain this improvement in performance, we linearly combined the results of the different algorithms with fine-tuned coefficients. We also briefly describe our most relevant and instructive experiments.

Additional Key Words and Phrases: Recommender systems, Top Popular, Collaborative Filtering, Content-Based Filtering, ALS, SVD

ACM Reference format:
Gerlando Savio Scibetta and Matteo Pozzi. 2017. Recommender System Algorithms for Implicit Job Dataset. *ACM Trans. Web* 1, 1, Article 1 (February 2017), 4 pages.
DOI: 0000001.0000001

1 INTRODUCTION

This is our report for the Kaggle competition relative to the 2016/2017 course of Recommender Systems.

1.1 How we evaluate the MAP locally

To evaluate the performance of our algorithms we split our data into a training set and a test test. To calculate the MAP@5 of the submissions generated by our models we used the implementation of this metric that can be found in the Kaggle documentation. Once we trained our models with the training data, they generate submissions that can be evaluated on the test set. The models which resulted in the highest local map were then re-trained to include all the available data.

1.2 How we recommend items to each user

Our goal is to recommend five unseen, relevant items to each user, whether our training set includes their past interactions with the application or not. To achieve this goal, we assign a different score to each item in our data set for each user, which we will call $r_{u,i}$. Once we estimate this score, we simply recommend to any user u, the five i items with the highest score.

2 FINAL ALGORITHM

We first explain the algorithm for the subset of the target users for which we have past interactions in our records. For users who have not yet interacted with our application, the relevant items are

This work is supported by the National Science Foundation, under grant CNS-0435060, grant CCR-0325197 and grant EN-CS-0329609.
Author's addresses: G. S. Scibetta and M. Pozzi, DEIB, Politecnico di Milano.
 1559-1131/2017/2-ART1 $15.00
DOI: 0000001.0000001

estimated with a different method, which we will explain later.

Our final algorithm is an hybrid algorithm which combines collaborative filtering and content-based filtering techniques to achieve a better Mean Average Precision. More precisely, the $r_{u,i}$ score is calculated as a linear combination of the scores obtained by different algorithms. In fact, each of the algorithm we used outputs a user score matrix R with dimensions $M \times N$ where M is the number of users and N is the number of items.

The result is a weighted sum of these matrices. It is very important to note that in order to make the scores of each algorithm comparable we normalized each row (corresponding to the predicted scores of a user) by dividing it by its maximum value, so that we obtain only numbers between 0 and 1. Specifically, we used this weighted sum:

$$R_{HYB} = 0.65 \cdot R_{CBF-Tags} + 0.35 \cdot R_{CBF-Title} + 0.9 \cdot R_{ALS} + 0.1 \cdot R_{CF-TopPop} + 0.1 \cdot R_{SVD-Time} + 0.1 \cdot R_{CF-Time}$$

We now proceed to explain each term and abbreviation in this formula:

2.1 CBF-ItemItem-tags — CBF-items

The first two matrices are obtained with a Content Based Filtering approach, a K-Nearest Neighbor algorithm over the items where the distance metric used is the Jaccard. The reference for this algorithm are the lecture notes of the course. We found that the most important features that needed to be compared between two items were their tags and titles. Specifically, to compute the distance matrix between items we generated a Item Content Matrix (ICM) where each row represents an item and each column represents either a single tag or a single title for all the different tags and titles that appear in the dataset. (Only tags that appear in at least 2 different items are considered). The element a(i,t) of this matrix is 1 if the item I has the tag/title t in its description, 0 if it doesnfit. From this it is easy to calculate the Jaccard Similarity Matrix using the known method.

2.2 Collaborative Filtering

- **Alternative Least Squared (ALS):** This algorithm takes our User Rating Matrix (MxN) and factorizes it in two smaller matrices: a matrix U (MxK) with the latent user feature vectors for each user and a matrix V (KxN) with the latent item feature vectors for each item. These two matrices are used to compute the values of each pair of items of the new similarity matrix.
- **CF-TopPopular:** This is obtained by a simple K-NN CF User-Based technique. The similarity time we use is the cosine. To improve this technique, we take into account that our data set includes items that result in a high rating score for a user, but that are very unpopular and have received only very few hits. To take into account item popularity we multiply each $r_{u,i}$ score in the result by a popularity weight calculated based on the number of clicks that items has received from all the users:

$$w(clicks) = 1 - \frac{1}{\sqrt{clicks}} \tag{1}$$

  where *clicks* is the number of clicks on the item.
- **SVD-Time:** This matrix is computed with an SVD technique where the initial User Rating Matrix (our data set) is not made up of only 0 or 1 elements but each element is a number between 0 and 1, scaled for the time that has passed from the interaction (based on the timestamp provided in the data). SVD is very similar to ALS, it decomposes our URM ($M \times N$) matrix into two orthogonal matrices U ($M \times K$), Vt ($K \times N$) and a diagonal matrix

of singular values S ($K \times K$). Then, like in ALS, these three matrices are used to compute the values of all pair of items of the similarity matrix in output.
- **CF-Time:** This is another K-NN CF User-Based technique but for which the initial URM is as the one in the SVD-Time technique.

The **weights** were estimated by trial and error and comparing the resulting local map. Another approach we tried was an heuristic search of the maximum local map with Nelder-Mead optimization, for which there exist a good implementation included in the scipy python library. However, the time necessary to perform this optimization did not result in a score that was improved enough to justify the effort, compared with just trying different weights by hand.

As we anticipated above, these methods are not applicable to users with no past interactions. For them, we extract recommendations from a different matrix R:

$$R_{HYB2} = 0.3 \cdot R_{CBF-UserUser-jobroles} + 0.2 \cdot R_{CBF-UserUser-edu\_fieldofstudies} + 0.1 \cdot R_{TopPopular}$$

- $R_{CBF-UserUser}$: these are K-NN methods where the $r_{u,i}$ scores for a user u are estimated on the base of scores computed for its most similar other users.
- $R_{TopPopular}$: is an R matrix calculated with a score based on the overall number of clicks received by an item.

3 HOW TO REPRODUCE OUR EXPERIMENTS/ALGORITHM IN THE CODE

First of all, it is very important to note that our code was meant to be as modular as possible. Each of the single model runs as a separate Python script, completely independent from the others. This reflects our need to explore different techniques and a new programming language, and making fast changes, without having to redo any of the work. Our code is not meant as a connected library. However, one advantages of this is that the usage of our code is really straight forward: to compute the score matrix for an algorithm you just need to run (make sure your interpreter includes all of the required library, in particular some scripts may require a license of Graphlab to run - although most wonfit.) The resulting matrix is then stored into local memory as a file with .npz extension (for more, see: http://stackoverflow.com/a/8980156/4638586). You can then load these files into the script *submission_output.py* or *local_evaluation.py* in order to produce a .csv submission with all recommendations for each target users, or to locally evaluate the map, respectively. Further instructions are included in the comments of the code.

4 EXPERIMENTS

- **Bayesian Personalized Ranking (BPR):** having a data set with implicit feedback, the first experiment we have implemented was Bayesian Personalized Ranking applied to Matrix Factorization. This method is optimized for ranking and we use it with MF to predict a personalized ranking of items for each user. It creates user specific pairwise preferences between two items i and j, where a user u prefers item i over item j when u has interacted with i but not with j.
  The problem with our implementation was that our sampling was too slow and to reach good results it needs too much time. Then we tried to use the code given by the teaching assistant, but in this case we have bad results in terms of MAP: we reach a better result with other algorithms.
  At the end, we have tried to use some library from the web, like MyMediaLite. Also in this case, we havenfit reached good results on our MAP.

- **BPR-MF + ALS:** given the bad results reached using BPR with MF, we have tried to experiment this method in cascade with ALS, given that with this last approach we had very good results. What we have done was simply take the resulting output matrix from BPR and give it to ALS. This implementation was quite fast, but the result was very bad, more or less the same as top popular.
- **Graphlab - MF and CF:** one of the most promising experiment results were given by the popular Graphlab non-free library. However, in the final algorithm only traces of the results obtained with Graphlab are present. We tried many of the provided recommendations models. The most relevant were the Matrix Factorization and the Collaborative Filtering models. The evaluation of its performance then proved very useful as a baseline to our own custom algorithms, which we had to code because of the very limited flexibility of the library.
- **Other Python libraries:** our initial intention was to code by hand newer and different algorithms such as ALS and SVD, but the execution was very slow, so we have decided to use some libraries to speed up the computation. For ALS we have used the package *implicit*, which provides a fast python implementation of the algorithm. It automatically computes the users and items vectors with which then we creates the similarity matrix between items. For SVD we have two different packages: *sparsesvd* and *svds* from scipy.sparse.linalg. After trying both them, the results and the computing speed were quite similar, so we decided to use svds which was already included in Anaconda. It decompose the URM matrix in two matrices S and Vt and the vector of singular values s, then we transform this vector into a matrix S. Also in this approach, using the three matrices, we compute the similarity matrix between items, but the results were not so good as we expected.

REFERENCES

[1] Yifan Hu, Yehuda Koren, Chris Volinsky. *Collaborative Filtering for Implicit Feedback Datasets*. IEEE International Conference on Data Mining (ICDM 2008), IEEE, 2008 Addison-Wesley, Reading, Massachusetts, 1993.

[2] Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, Lars Schmidt-Thieme. *BPR: Bayesian personalized ranking from implicit feedback*. Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence, 2009, Montreal, Quebec, Canada.

[3] Jesse Steinweg-Woods. *A Gentle Introduction to Recommender Systems with Implicit Feedback*.