Recommending systems on implicit feedback data: a comparison between Collaborative Filtering and NN-based approaches

Jordy Dal Corso

`https://github.com/jdalcorso/recommending_nn`

Introduction

We can find recommending systems all around us during our daily life: for example at the supermarket or on the internet while watching videos or booking for a holiday trip. Recommending systems exploit user preferences and feedback in order to make ad-hoc suggestions for products.

In general when we want to create a recommending system we need a set of users, a set of products (or *items*) and a list of interactions between them. An interaction can usually be represented by a couple user-item and an associated value, which can be a vote (or *rating*), but also the watching time of a video product, or simply whether or not an interaction happened.

After gathering all the data, the creation of a recommending system proceed according to its main purpose: suggesting items to users who would probably like them. In order accomplish this task, different approaches has been exploited during the years. Most of them lies around the concept of *similarity*. Suggesting similar items to an user seems natural: for example a Netflix user who watched a big variety of anime series would probably like another one of them. In an almost analogue way we can also suggest an item to similar users: Jack and Jill seems to have the same tastes according to their ratings and Jack watched Titanic, why don't we suggest Titanic to Jill too?

The most famous approach for the creation of a recommending systems is *Collaborative Filtering* (CF). It exploits similarity between users and items using linear combinations in order to make suggestions and it will be introduced in section 3.1.

The main purpose of this report is to introduce and test another class of interesting approaches, which are based on neural networks (NNs): the main idea of these methods is to discover some hidden (or *latent*) features of users and items and combine them using the learning power of NNs, these techniques will be introduced in sections 3.2 and 3.3 and we will assume a basic knowledge in NNs theory.

All these methods will be trained and tested on a dataset where interactions between users and items are represented by binary values: this is a kind of implicit feedback that will be discussed in section 1. The dataset itself will then be introduced in section 2, along with the metric used to evaluate results, which are drawn and discussed in the final section.

1 Implicit feedback

The difference between explicit and implicit feedback is crucial in the field of recommending systems. In this report we will train our models and analyse results over an implicit feedback dataset, but let's start by introducing the explicit feedback. An *explicit* feedback corresponds to a direct opinion of an user toward an item: this is usually represented by a rating, which is usually a numerical value. Some examples of explicit feedback are likes and dislikes on Youtube videos or emoticons ratings on Facebook posts.

On the other hand *implicit* feedback is something that can be gained by observing user's behaviour without the need of a direct rating. Some examples are watching time of a video, whether or not an interaction occurred or permanence time over a thumbnail.

Such interactions allows to create an interaction matrix with dimensions $\#users \times \#items$ whose entry (i, j) represents the rating (or level of interaction in the case of implicit feedback) of user i with respect to item j.

While explicit feedback data span over different scales, usually implicit feedback data are represented by numbers between 0 and 1 where 1 represents a full interaction (e.g. a video has been watched for its full length) while 0 represents no interaction at all. Notice that 0 does not necessarily mean dislike, but can also represent the fact that an user doesn't know an item or is not able to retrieve it (maybe due to pay-per- view or restrictions).

In the next sections a description of the implicit dataset used in this report is performed and then three models that can deal with implicit feedback datasets are introduced, with a particular focus on the neural network-based ones.

2 Dataset

The dataset used in this report can be found in [3]. It is the Movielens 1M dataset containing over 1 million ratings given by 6040 users to 3706 films.

The dataset consists into a list (text file) of interactions and each interaction consists in an user, an item, a rating from 1 to 5 and a timestamp. Each user is represented by a number (*userid*) and the same is for films/items (*itemid*).

For this report the aim was to convert this dataset into an implicit one. It was relatively easy to do since it was enough to transform every non-zero rating into the value 1. In this way ratings has become binary values where 1 means an interaction occurred and 0 means no interaction. With such a dataset we don't really know whether an user liked an item, but we only know there had been an interaction between them, then this turns out to be an implicit dataset. This seems to lead to an oversimplification of the dataset but notice that knowing that an interaction occurred means knowing there is some interest and usually an explicit rating is way harder to get than a binary implicit information.

Finally the list of interactions allowed to create the interaction matrix with dimensions 6040×3706. Every row of the matrix is a vector representing interactions or non-interactions between the user and each of the items.

3 Setup

As anticipated, 3 approaches are developed and compared in this report. The implementation has been carried out using Python, in particular Jupyter notebooks. See the Github link for further information regarding the code.

All the notebooks has been created and used on a Macbook Pro 2019, 1,4 GHz Intel Core i5 quad-core processor, 16 GB 2133 MHz LPDDR3 RAM.

3.1 Collaborative Filtering

Collaborative Filtering (shortly CF) is the most famous approach to build a recommending system. The implementation largely follows [6] and the lecture notes and it is based on neighborhood between different items (i.e. *item-item CF*). Another kind of CF is carried out by [5] and is based on latent factors but in this report we will only focus on the "standard" neighborhood-based one in order to compare it with the NN-based ones.

In general the implementation of a neighborhood-based CF only need as input the k nearest neighbors of each item to consider and the matrix with dimensions $\#users \times \#items$ containing the ratings of users toward items. In our case matrix entries contains binary values, as we have seen before. The workflow for the creation of a neighborhood-based CF is as follows:

1. Loading the dataset (ratings matrix)
2. Finding the k nearest neighbors of each item and their distances
3. Making predictions using the formula:

$$r_{ui} = \frac{\sum_{j \in N(i)} s_{ij} \cdot r_{uj}}{\sum_{j \in N(i)} s_{ij}} \tag{1}$$

In the latter r_{ui} refers to the predicted rating of user u to item i, $N(i)$ refers to the set of neighbors of item i and s_{ij} is the similarity between items i and j. Then the predicted rating is a weighted normalized sum of the ratings of the neighbors and the weights are their similarities with respect to item i.

The implementation of this method has been carried out using the library in [2]. It allows to create neighborhood-based recommending system as well as many other (more developed) techniques. Library is still updated nowadays and it actually seems the best and fastest way to deal with implicit feedback datasets with Python.

We applied CF with two different similarity metrics:

- **Euclidean**, carried out by *ItemItemRecommender* class. Euclidean distance between two N-dimensional vectors is calculated using the formula:

$$d_{xy} = \sqrt{\sum_{k=1}^{N} (x_k - y_k)^2}$$

- **Cosine**, carried out by *CosineRecommender* class. The cosine distance between two vectors is the cosine of the angle between them, which can be obtained dividing their dot product by the product of their norms:

$$d_{xy} = \frac{x \cdot y}{||x||||y||}$$

Notice that similarity is strictly related to distance and they allows to find neighbors. In general we can switch from similarity to distance using a decreasing function (with some assumptions), for example in the case of cosine metric we have:

$$sim_{cos} = 1 - dist_{cos}$$

Another way to calculate similarities and so neighborhoods is by using *sklearn.neighbors.NearestNeighbors* but somehow *implicit* library in [2] manage to do everything (neighborhood and model fitting on ratings) faster and within one line of code.

For each of the two similarity metrics we tested a different number of neighbors N, namely

$$N \in \{10, 20, 50, 100, 200\}$$

3.2 Multi-layer perceptron

Multi-layer perceptron (MLP) is an artificial neural network whose structure is divided in *layers* and each layer consists in different *units.* with associated weights. MLP allows to find a function f, which is a composition of functions that allows to predict a response (in our case the *rating*) given an input, which in our case is a couple user-item. In order to find the best parameters (i.e. weights) for the composition, gradient descent algorithms are usually used, together with a smart calculation of the gradient also known as *backpropagation.*

Without going too deep into how the model learns its best parameters, we provide some information on how this NN-based model works. The general idea is to transform each user and item into an M-dimensional vector, where m is also known as the dimension of the *latent space.* Each entry of the latent vector of a movie, in our case, can represent its features, like comedy, whether or not it is an animation movie, presence of blood and so on. Finally the information of each user and item, which are encapsulated in their latent vectors are combined using a neural network in order to provide an output, which is the predicted rating (i.e. interaction in the case of implicit dataset) of an user toward an item.

Notice that neural networks not only provide a way to learn parameters to get the best output (rating) out of a tuple $(user, item)$ but also provide a way to learn embedding/latent vectors: feeding the network with enough training tuples allows the model to learn the best fitting latent vectors for each user/item.

For simplicity purpose, the latent space of users and items is the same, but a further analysis can be carried on in future works.

We can then summarize the structure of the MLP, ad so its layers as follows:

- **Input layer** This is a tuple $(user, item) \in \mathbb{N} \times \mathbb{N}$
- **Embedding layer** Each entry of the input tuple is converted into a vector whose dimension is m.
- **Hidden layers** Embedding vectors are concatenated and fed into a standard feed-forward NN with fully connected layers
- **Output layer** The final layer consists in only one unit $r_{ui} \in (0, 1)$ which is the predicted output/rating given as input the tuple (u, i)

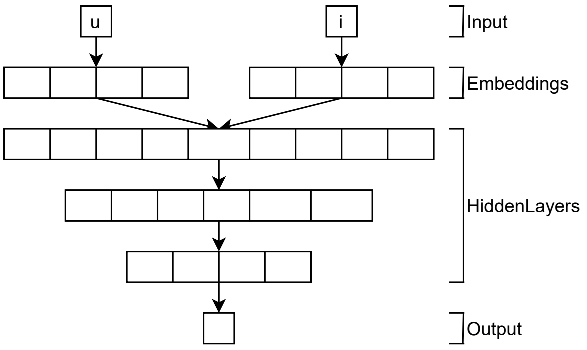

Figure 1: Structure of MLP network

Output 0 means the model predicts no interaction between (u, i) while output 1 means interaction. Values in $(0, 1)$ can be interpreted as probabilities of interaction.

Networks has been created using *keras.Sequential.* Different architectures for the network has been tested. The architecture of each network has been read and parsed by the scripts using a text file named *MLP_architecture.txt.* The file contains n natural numbers which represents the number of units for each layer: we tested different network structures with $n = 3$.

The name of each network is $MLPm$ following [4], where m is the dimension of the latent space, so the dimension of the first hidden layer (which is the concatenation of two latent vectors) divided by 2. The next layers are obtained dividing by 2 the dimension of the previous one and the final layer has dimension 1.

Each network has been trained using all the interactions in the dataset (1 million) plus 4 negative interactions for each positive interaction. With the term *negative interaction* we refer to zeroes of the interactions matrix, so no interaction. Negative interactions are used because the model need also to learn when to predict no interaction, and the ratio between positive and negative training interactions, which is 1:4 has been chosen due to literature (see [4]). For the time elapsed during training we can refer to the table below:

MLP-	4	8	16	32	64
Time(s)	296	309	350	404	564

Notice that around $180s$ for each model are used to create the training set at each epoch (with both positive and negative interactions). As a matter of example, training time per epoch was $37s$ for GMF-64 and $22s$ for GMF-32. Each model has been trained for 10 epochs over the same positive interactions (all of them) and different negative interactions at each epoch.

Finally we tested different structures and so different latent space dimensions m, namely

$$m \in \{4, 8, 16, 32, 64\}$$

and the evaluation protocol is better explained in section 4.

3.3 Generalized Matrix Factorization

Generalized matrix factorization (GMF) is a particular case of MLP developed and tested in [4]. Basically it is a MLP composed only by one layer. The only layer acts as a weighted dot product between the two embedding vectors of input user and item and weights are learned during training.

Standard matrix factorization (MF) in literature consists into a dot product between latent vectors in order to produce the final output/rating. GMF is somehow a generalization of this technique since it learns the best weights that multiply each of the entries of the latent vectors in order to obtain a weighted dot product. To better explain this, we call v_u and v_i respectively the latent vectors associated to user u and item i and r_{ui} the prediction of each of the models. Then MF consists in:

$$r_{ui} =< v_u, v_i >$$

while GMF consists in:

$$r_{ui} =< w_u v_u, w_i v_i >$$

and weights w_u, w_i are learned by the network during training.

Simple MF can also be obtained using *keras* by using only a multiplication layer with fixed weights consisting in vectors whose each entry is 1 and no training process is needed.

In the notebooks, a parameter regulates whether the model is an MLP or a GMF. In both cases a sequential NN is created but the parameter tells whether the embedding layers are concatenated or multiplied and how to read the file *MLP_ architecture.txt*.

The training procedure is the same as for MLP, using both positive and negative interactions. The training time is shown below:

GMF-	4	8	16	32	64
Time(s)	263	272	325	351	401

As in MLP, around 180 seconds are used to build the training set and for the flat training time of each epoch we cite GMF-32 with $18s$ per epoch and GMF-64 with $24s$ per epoch. This is slightly faster than MLP because actually less parameters needs to be trained, but as we will see later this will lead to worse performances.

Different dimensions for the latent space has been tested and they are the same as in the case of MLP.

4 Metrics and results

The metric used to evaluate performances is Hit Ratio (HR). Given an user u and an item i, supposing interaction (u, i) occurred, we have an *hit* if the model correctly detects/predicts an interaction between the two so given (u, i) as input, the output is 1. This kind of hit is very restrictive since we never get a prediction which is exactly 1. We can deal with this problem using a threshold but in the literature (see [4]) the idea is to detect the interaction among some other non-interactions.

For each user u, a test (positive) interaction, which consists in a non-zero entry of the interactions matrix, is used together with 99 negative interactions (no interaction occurred). The model calculates its prediction for each of these interactions, which consists in couples $(u, i) \in \mathbb{N} \times \mathbb{N}$. Then each interaction is associated to its predicted rating and tuples are rearranged in descending order by their rating. Finally if the positive interaction is in the top-10 of the final standings we have an hit.

Both the number of negative samples n and the value k for the final top-k can be tuned but in this report we used $n = 99$ and $k = 10$.

Some interactions are taken away from the training set (which consists in the interactions matrix) and used as positive test interactions. In particular one interaction per user is used as test set (to evaluate the model on that user). The 99 negative interactions are randomly selected for each user.

Then the evaluation is carried on by the function *evaluate-user-k()*. In this way an user can be an hit or not, and finally the HR is calculated dividing the number of hits by the total number of users.

4.1 CF results

Results for CF methods are shown in the next figure. Apparently for both Cosine and Euclidean Neighborhood-based CF the HR is under 0.5 with a peak of 0.451 for Cosine CF-50.

Cosine CF outperforms the Euclidean CF at all numbers of neighbors: the Cosine metric seems to be one of the best way to deal with recommending systems since the magnitude of latent/embedding vectors doesn't really matter as much as their orientation.

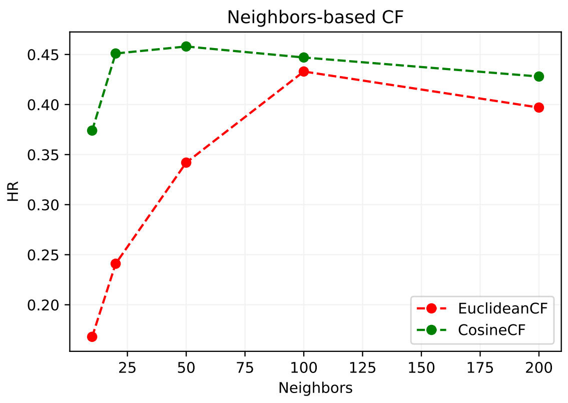

Figure 2: HR results for CF

We can also notice that a good number of neighbors lies around 50 to 100. Higher numbers lead to overfitting while lower numbers are simply not enough to make good predictions.

Since results are lower than expected, there are probably better ways to deal with CF and we can see them for example in [5]. In the latter a [**confidence**] parameter is introduced in CF since dealing with implicit datasets lead to uncertainty in the interactions: an entry of the interaction matrix can be 1 (i.e. in a movie has been watched, in the case of Movielens) but maybe an user is just staying on the channel of the previously watched movie.

This is also why we decided to introduce NN-based models, which results are shown below. NNs don't address the confidence problem, but actually consists in a completely different model with respect of CF. Neighborhood-based methods nonetheless can be trained much faster than NN-based ones and their parameters can actually be adjusted to get better results, as we can see in [4], but still in the article MLP and GMF outperforms classical methods.

4.2 MLP and GMF results

As we can see from the plot below, results for NN-based methods lies over 0.5 with a peak of 0.705 for MLP-32 and 0.697 for GMF-32. MLP outperforms GMF almost at every latent space dimension except for the lower one: actually MLP has a more complex architecture and seems to fit better so the results are as expected. On the other hand, MLP training time is higher than the one of GMF as we have seen before.

Notice that we only tested MLP with 3 layers: benefits of deep learning and so benefits of a deeper network with more than 3 layers has been tested for example in [4] and leads to even better results. This leads to think about MLP as a more versatile and precise model, provided enough computational speed.

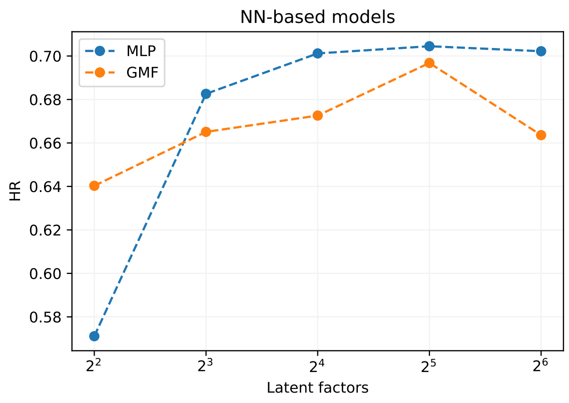

Figure 3: HR results for MLP and GMF

Overfitting occurs when dealing with NN models too: latent spaces with higher dimension than 32 lead to poor results. Talking about the specific case of Movielens, in an high dimensional latent space latent vectors become over-specialized on every movie and fails to generalize and finding common features between different items.

In the final plot below we summarize the comparison between CF and NN-based models. The best CF models results are shown and we can see they are poor with respect to the NN-based models. A profound difference between the two approaches can be found in training time because while training time of NN-based approaches lies around hundreds of seconds in our setup, training times of CF using *implicit* library are in the order of 10 seconds.

In any case results lead to think about NN-approaches as a valid alternative to classical methods.

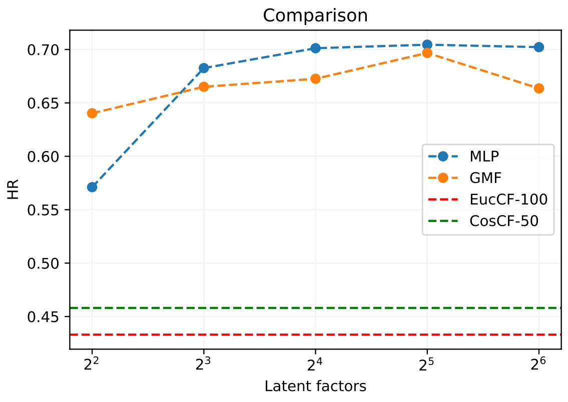

Figure 4: Comparison between NN-based and CF

The potential of MLP to be one of the state-of-art approach to deal with recommending systems has been only slightly enlightened. A further analysis can be carried on over different datasets and metrics and NN-based recommending systems have already been implemented in relevant companies like Youtube, as we can see in [1]. Further developments can also be carried on by combining results of GMF and MLP, as described in [4]. The field of neural networks is wide and growing and we're enthusiast for having introduced its application in recommending systems, which nowadays is an hot field too.

References

[1] Paul Covington, Jay Adams, and Emre Sargin. "Deep Neural Networks for YouTube Recommendations". eng. In: RecSys '16 (2016), pp. 191–198.

[2] Ben Fredrickson. *implicit*. `https://github.com/benfred/implicit`.

[3] Grouplens. *Movielens 1M*. `https://grouplens.org/datasets/movielens/`.

[4] Xiangnan He et al. "Neural Collaborative Filtering". In: *CoRR* abs/1708.05031 (2017). arXiv: `1708.05031`. URL: `http://arxiv.org/abs/1708.05031`.

[5] Yifan Hu, Y Koren, and C Volinsky. "Collaborative Filtering for Implicit Feedback Datasets". eng. In: (2008), pp. 263–272. ISSN: 1550-4786.

[6] Anand Rajaraman. "Mining of massive datasets / Anand Rajaraman, Jeffrey David Ullman". eng. In: (2012).

[7] Badrul Sarwar et al. "Item-Based Collaborative Filtering Recommendation Algorithms". In: WWW '01 (2001), 285–295. DOI: `10.1145/371920.372071`. URL: `https://doi.org/10.1145/371920.372071`.