Report

Recommender Systems

I Introduction

The aim of this task was to use data from Flickr to recommend photos to a set of given users. In order to so, I completed the following steps:

- Building, evaluating and comparing 3 different basic Neural Networks models: Matrix Factorization (MF), Generalized Matrix Factorization (GMF) and Multi-Layer Perceptron (MLP). After comparing their performances, I decided to move on with MF for further tuning.
- Having decided to continue with MF, I constructed a new model on top of this one using the pre-trained users and items data: MF_UI (UI for User-Item) and another one using all the extra data: users, items, and social links data (MF_UI_LINKS).
- I added biases to the chosen model from the previous step (MF_UI, as we will see) and played around with a hyperparameter I introduced myself: negative_ratio, which refers to the number of negative samples over the positive ones in the training data (negative samples being people not interacting with an image, which is synthetically added to the data and positive samples being people interacting, which is information given by the original data).
- I chose the best-performing model from the previous step and tuned the hyperparameter weight of decay, which is responsible for performing L2 regularization.
- Finally, I used the best-performing model of all to make predictions on the test data.

II Methodology

Basic Models

As stated in the introduction, firstly I presented the very basic version of the different alternatives, in order to understand which would be the one that could potentially perform better. The basic alternatives that I tried were:

- Matrix Factorization (MF) Model that consists of an embedding for the users and the items and achieves the prediction by using a product between these 2.
- Generalized Matrix Factorization (GMF) Model in which an extra layer is added to model the interaction between user and item.
- Multi-Layer Perceptron (MLP) that consists of three types of layers: input, output and hidden layers, that can be useful to model any non-linear relationship between users and items.

I believe it is important to highlight here that the training data only had positive instances, meaning that there was only data about users and items that had interacted. There was no data about items that users had not interacted with. I considered it was important to include negative samples among our training data, so that our model could really learn what the user likes and rank it higher for instance than items he dislikes or has not interacted with. To try our basic models, I started with 5 negative samples per every item the user had interacted with. Nevertheless, this N became a hyperparameter that I was able to tune afterwards and which ended up being pretty relevant in the final performance.

Performance of Basic Models
After training and validating all of our 3 models I decided to move on with MF which was the one that yielded the highest performance. In addition, it was the simplest model, which is always a good thing.

Models with pre-trained data

I built here 2 different models similar in structure to the basic MF, but including pre-trained data: the first one with the pre-trained data for users and items and the second one containing users, items and social links data.

First model with pre-trained data: users and items
By taking into account that in MF the prediction is finally achieved by U*V, where U and V are the embeddings of user and item respectively, I included the pre-trained data in the following way:

$$(U + pre\_trained\_user) * (V + pre\_trained\_item).$$

By working out the cross-products, I ended up with the following expression that is the one that ultimately produces the predictions:

$$(U * V + U * pre\_trained\_item + pre\_trained\_user * V + pre\_trained\_user * pre\_trained\_item)$$

It is important to remark that the weights of the pre-trained embeddings were not updated during the optimization, because they were supposed to have been already adequately trained.

Second model with pre-trained data: users, items and social link data
Social link data is strictly referred to the users, so our "user side" of the product had now 3 components:

$$(U + pre\_trained\_user + social\_links) * (V + pre\_trained\_item)$$

Which yielded the following after working out the cross-products

$$U * V + U * pre\_trained\_item + pre\_trained\_user * V + pre\_trained\_user * pre\_trained\_item + social\_links * V + social\_links * pre\_trained\_item$$

I finally factored further the sparse version of the social link matrix into 2 matrices W and H and the resulting expression (which I am not including here because it is just too long) is what I used for the final prediction. Just as in the previous case, user, item and social link data’s weights were not updated during training.

I believe also to be important to highlight that from this point onwards I started to train the models using decreasing learning rates as the epochs increased and it seemed to yield good results.

Results for the models including pre-trained data
The results of both models were considerably better than the ones obtained using the basic models. Both models containing pre-trained data yielded very similar results but I decided to continue with the one including only users and items because it was simpler.

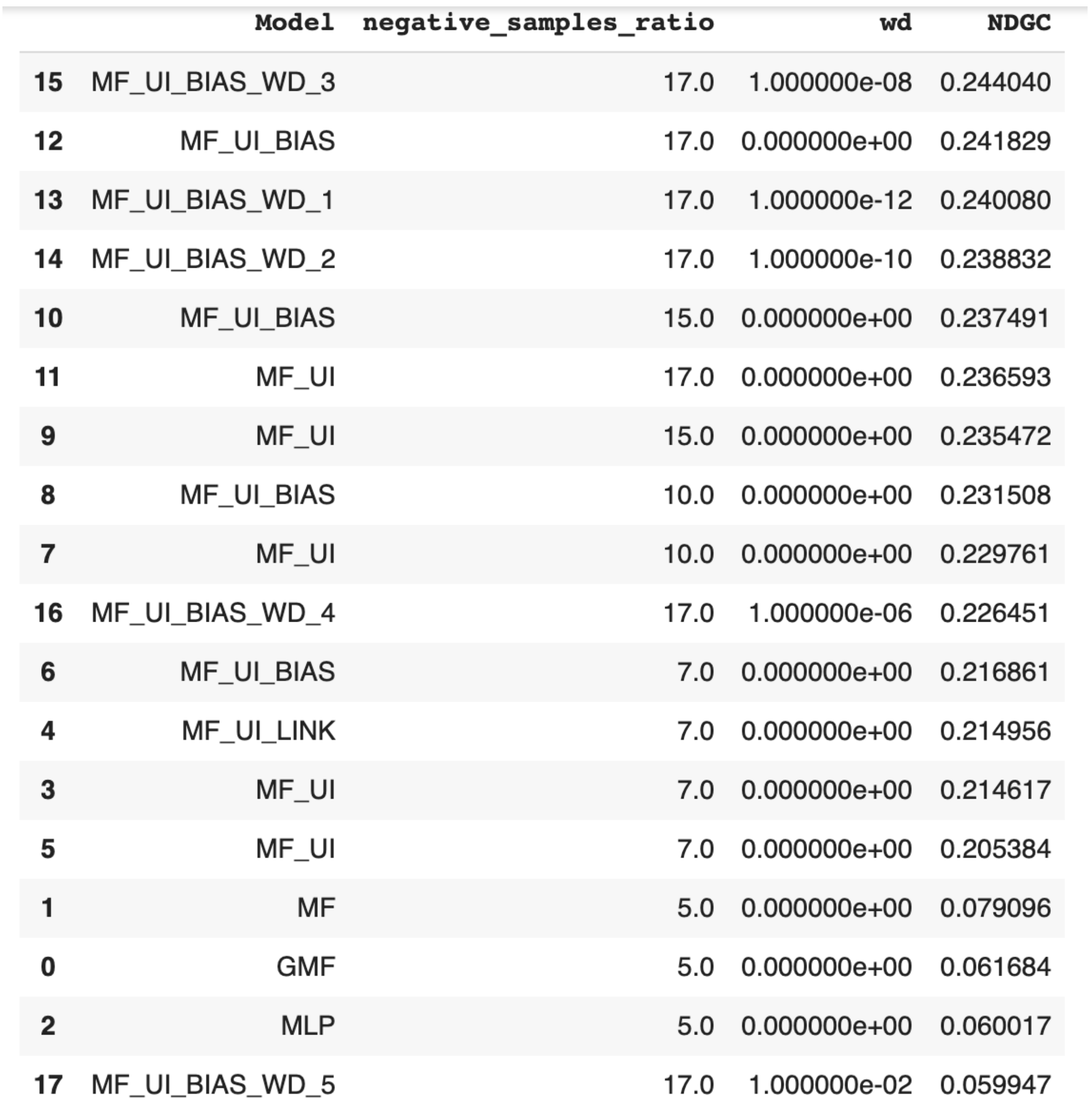
MF model with pre-trained UI and biases
I built another model adding biases to the one selected in the previous step. The biases correspond to the users and the items. I believe it was a good idea to represent this in our model because there are always users that will have a tendency to like more images than others, as well as images that will be naturally more liked than others, independently of the interaction between the 2.

Negative_ratio hyperparameter tuning and comparison between models
In order to compare the MF model with pre-trained data with the same one but including the biases, I used different values for the negative_ratio hyperparameter I introduced. The results for the same value of negative_ratio were in all cases slightly better in the model with biases and the performances of the pair of models increased as we increased the value of negative_ratio, meaning that our models worked better when we included more images that our users had not interacted with in our training data.
So, the selected model to continue with was the one with biases and the highest negative_ratio I was able to try: 17. Unfortunately, I just got to try up to a negative_ratio of 17 because the system started to crash for higher values (every time we increase this value we are considerably increasing the size of our data set, being 1972744 rows for negative_ratio = 17). Whenever I worked with batches, it was extremely time-consuming and even the performances got considerably worse.
As I systematically got higher scores when increasing this parameter (at least until 17), I believe that if I had the computational power to keep increasing it, I would have reached even higher performances.

Weight of decay tuning
The weight of decay is an implementation of the L2 regularization so it can be helpful to prevent overfitting. With larger values for wd we are imposing a larger penalty to the model, reducing the size of its coefficients. With smaller values of wd we would be allowing the model to be more flexible, imposing no penalty at all when wd=0 (which had been the case of the models we have tried until this point).
I tried the model I selected from the previous step with multiple values for the weight of decay parameter and ended up having the best performance for a value of 1e-8 which is actually a value that is really close to 0. This means that the model is still allowed a considerable flexibility.

III Final Results

The following table summarizes the results obtained for all of the models I have tried, ranked by performance:



	Model	negative_samples_ratio	wd	NDGC
15	MF_UI_BIAS_WD_3	17.0	1.000000e-08	0.244040
12	MF_UI_BIAS	17.0	0.000000e+00	0.241829
13	MF_UI_BIAS_WD_1	17.0	1.000000e-12	0.240080
14	MF_UI_BIAS_WD_2	17.0	1.000000e-10	0.238832
10	MF_UI_BIAS	15.0	0.000000e+00	0.237491
11	MF_UI	17.0	0.000000e+00	0.236593
9	MF_UI	15.0	0.000000e+00	0.235472
8	MF_UI_BIAS	10.0	0.000000e+00	0.231508
7	MF_UI	10.0	0.000000e+00	0.229761
16	MF_UI_BIAS_WD_4	17.0	1.000000e-06	0.226451
6	MF_UI_BIAS	7.0	0.000000e+00	0.216861
4	MF_UI_LINK	7.0	0.000000e+00	0.214956
3	MF_UI	7.0	0.000000e+00	0.214617
5	MF_UI	7.0	0.000000e+00	0.205384
1	MF	5.0	0.000000e+00	0.079096
0	GMF	5.0	0.000000e+00	0.061684
2	MLP	5.0	0.000000e+00	0.060017
17	MF_UI_BIAS_WD_5	17.0	1.000000e-02	0.059947

We can see how our best-performing model is the MF including the pre-trained users and items data, with a weight of decay = 1e-8 and the maximum negative_ratio I was able to try: 17.

IV Conclusions

Taking into account all that has been stated in the Methodology section and also observing the final results I was able to conclude the following:

- More complex does not necessarily mean better. In this case, while comparing the first 3 basic models I tried, I realized that the simplest one was yielding the highest performance, which was also really convenient because it was easier to further tune.

- The inclusion of the pre-trained data considerably boosted the performance of my model. I believe that the decision of not updating the weights of these embeddings was also a key factor to obtain a better performance (because these weights were supposed to have already been appropriately trained) and to save computational expenses as well (because no optimization is required for all this new data).
- Gradually decreasing the learning rate for the model to allow it to refine itself, making smaller steps towards the opposite direction of the gradient as the training instances increased also helped improve the performance.
- Including biases to my model boosted the performance as well. I consider this was due to the fact that the possibility of initial tendencies regarding the users and the items was now being captured by the model which allowed it to make better recommendations.
- The optimal value for the regularization parameter was different than 0, which denotes that the model needed to reduce its complexity to be able to avoid overfitting but it was also a small value which means that a great deal of flexibility was still being allowed. If we take a look at the table, for the largest value of weight of decay we have the worst performance of them all, which is clearly a sign of an over-simplified model that is excessively penalizing flexibility.
- The ratio of negative samples over positive ones became a key factor in the final performance. I believe this is because with the inclusion of more negative samples, the model can really learn what the user likes and rank it higher for instance than items he dislikes or has not interacted with. Nevertheless, as stated before, this came at a very high computational cost.