Machine Learning - Project 2
Recommendation System

Jeanne Fleury
Julien von Felten
Regis Croset

***Abstract*—Recommendation systems are more and more used nowadays. The Netflix Prize challenge, based on recommendation, moved every machine learning expert around the world. One always wants to create systems with the highest level of precision to recommend items to users. Building one with a satisfying level of accuracy and performance is a real challenge.**

I. Introduction

In this project, we will implement a recommendation system. The challenge is to find a suitable algorithm that is efficient and provides a good level of accuracy. In the context of this project, based on the provided data, we will try to predict what items a particular user will enjoy or not based on his ratings and the ratings of other users.

We will try and explain different models to achieve this goal, present results for all of those models, discuss those results and try to present the best model.

II. Models and Methods

A. Data Pre-Processing

The very first step is to process data. We were given a csv file with two columns. The first one contains IDs (in the form rX_cY, where X stands for a user ID and Y for an item ID). The second one contains the actual ratings, i.e., an integer between 1 and 5. From that, we build a matrix of size $max(X) \times max(Y)$, filled with the ratings, and 0 for non-rated items. $Max(X)$ is 10000 and $max(Y)$ is 1000.

B. Cosine Similarities

1) User-Based Collaborative Filtering: The first model we tried is a user-based collaborative filtering using cosine similarity, which is:

$$similarity = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{||\mathbf{A}|| \cdot ||\mathbf{B}||} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \cdot \sum_{i=1}^{n} B_i^2}$$

Based on that, we can find k similar users and compute a weighted average of deviation from neighbor's mean and adding it to user's mean rating. This leads to the formula:

$$p_{a,i} = \bar{r}_a + \frac{\sum_{u \in K} (r_{u,i} - \bar{r}_u) \times w_{a,u}}{\sum_{u \in K} w_{a,u}}$$

where $p_{a,i}$ if the prediction for target (or active) user a for item i, $w_{a,u}$ is the similarity between users a and u, and K is the neighborhood of most similar users.

The NearestNeighbors method from the sklearn Python library helped us to compute the k nearest neighbors. We then just had to compute mean and weights to find the prediction.

This model is quick and efficient to find the prediction for one particular item for one given user (≈ 1 second). It does not require any long training process. Just call the processing method with user and item IDs as parameters to get the prediction regarding the k nearest neighbors, given the initial matrix containing the ratings. However, this leads to a really slow computation if we have to populate a whole matrix (≈ 1 million entries in our case), since we must compute item per item. Plus, the computation time increases when the number of k neighbors considered increases.

2) Item-Based Collaborative Filtering: This approach is similar to the previous one, but by looking for similarities between pair of items instead of neighbors. This is done applying the following formula:

$$p_{a,i} = \frac{\sum_{j \in K} r_{a,j} w_{i,j}}{\sum_{j \in K} |w_{i,j}|}$$

where K is the neighborhood of most similar items rated by active user a, and $w_{i,j}$ is the similarity between items i and j.

The same caveats appears than the previous method: it is fast and efficient for one particular item of a given user, but it takes ages to fill in the whole matrix.

C. Matrix Factorization

The second model we tried is the matrix factorization. From the users and ratings given, we build a matrix of size U $\times$ D, where U is the number of users and D the number of items. We want to find K latent features and two matrices $\mathbf{P}$ (of size U $\times$ K) and $\mathbf{Q}$ (of size D $\times$ K) such that their product approximates $\mathbf{R}$:

$$\mathbf{R} \approx \mathbf{P} \times \mathbf{Q}^T = \hat{\mathbf{R}}$$

The prediction for user i and item j is then computed as follow[1]s:

$$\hat{r}_{ij} = p_i^T q_j = \sum_{k=1}^{K} p_{ik} q_{kj}$$

After finding the two matrices $\mathbf{P}$ and $\mathbf{Q}$, the values are floating numbers. The submission is accepting only integers

[1] https://github.com/chen0040/keras-recommender

between 1 and 5. We simply used the rounding method to get integer.

1) Matrix factorization using SGD: We first used SGD to factorize the matrix. For this model, we took inspiration from an online article[2]. To compute $\mathbf{P}$ and $\mathbf{Q}$, we initialize them with some values and then minimize the difference between their product and the original matrix $\mathbf{R}$, using stochastic gradient descent. The difference (error) to be minimized is:

$$e_{ij}^2 = (r_{ij} - \hat{r}_{ij})^2 = (r_{ij} - \sum_{k=1}^{K} p_{ik} q_k j)^2$$

The update rules for p_{ik} and q_{kj} can then be computed with the formula below, where the factor 2 of the derivative was included in γ. The learning rate is α (then γ for simplicity):

$$p'_{ik} = p_{ik} + \alpha \frac{\delta}{\delta p_{ik}} e_{ij}^2 = p_{ik} + \gamma e_{ij} q_{kj}$$

$$q'_{kj} = q_{kj} + \alpha \frac{\delta}{\delta q_{kj}} e_{ij}^2 = q_{kj} + \gamma e_{ij} p_{ik}$$

We introduce regularization to avoid overfitting (add some parameters λ to control the magnitudes of the vectors) and this leads to the new updates formulas:

$$p'_{ik} = p_{ik} + \gamma(e_{ij} q_{kj} - \lambda_p p_{ik})$$

$$q'_{kj} = q_{kj} + \gamma(e_{ij} p_{ik} - \lambda_q q_k j)$$

These update formulas are then iteratively applied to the vectors p and q, for a given number of iterations.

2) Matrix Factorization using ALS: After using SGD, we used the ALS method to factorize the matrix. To compute $\mathbf{P}$ and $\mathbf{Q}$, we first fill them with random numbers. Then, we use the algorithm as follow: minimize $\mathbf{P}$ with respect to $\mathbf{Q}$ and then minimize $\mathbf{Q}$ with respect to $\mathbf{P}$. We repeat those 2 steps until the loss function is below a threshold. To minimize both matrix, we need to update them using two equations:

$$\mathbf{P}^T = (\mathbf{Q}^T \mathbf{Q} + \lambda_u \mathbf{I}_k)^{-1} \mathbf{Q}^T \mathbf{R}$$

$$\mathbf{Q}^T = (\mathbf{P}^T \mathbf{P} + \lambda_f \mathbf{I}_k)^{-1} \mathbf{P}^T \mathbf{R}^T$$

The loss function we are using is the MSE. To compute the loss function, we use the equation:

$$L(\mathbf{P}, \mathbf{Q}) = \frac{1}{n} \sum_{\Omega} (r_{dn} - (\mathbf{P}\mathbf{Q}^T)_{dn})^2$$

where n is equal to the number of votes from the training set. The set Ω represents all the indices that is not zero in the R matrix.

To implement this ALS factorization, we took the lab 10 of the Machine Learning course as inspiration[3].

[2]http://www.albertauyeung.com/post/python-matrix-factorization/

[3]https://github.com/epfml/ML_course/tree/master/labs/ex10

D. Neural Network

1) Matrix Factorization with Neural Network: This model uses an artificial neural network for recommendation, or for estimating ratings. It uses embedding to learn complex non-linear relationships. We created different dimension of user and item embeddings, which is useful because the user dimension is larger than the item dimension. This model was implemented using the keras python library, which uses Tensorflow. We took inspiration of an online article[4] to implement this model.

Each input is embedded and flatten separately. Then a dropout layer is applied to avoid overfitting. The two *branches* (users and items) are then merged and go through a series of fully-connected dense layers before reaching the final activation layer (see fig. 1).

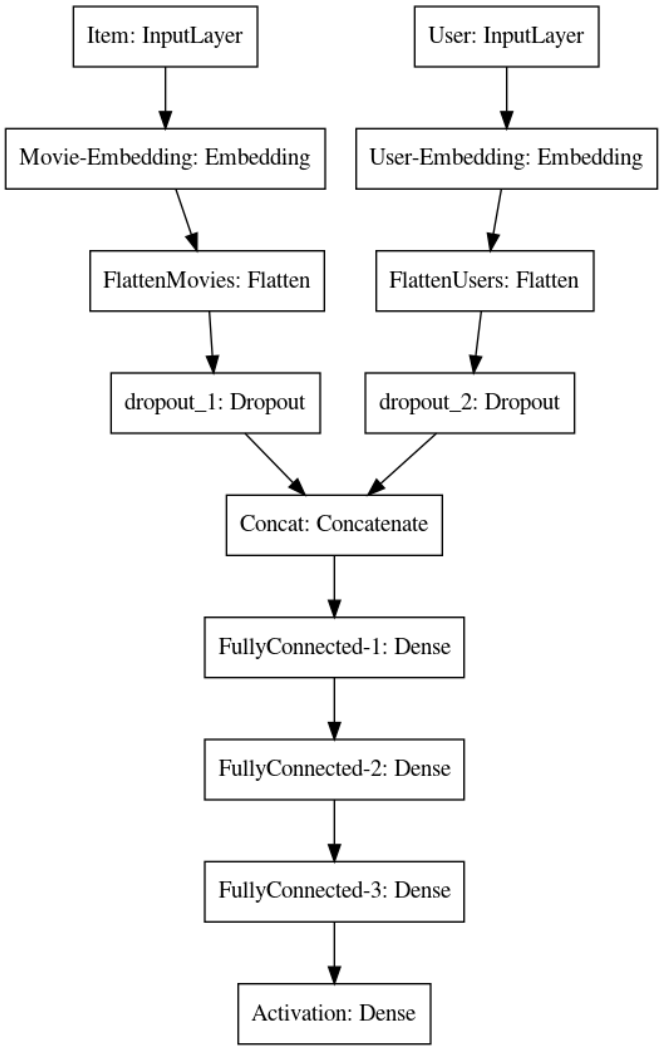

Fig. 1. Model for Neural Network Matrix Factorization

The dataset is then split into train and test subsets (80% and 20% of the whole dataset, respectively). The training is

[4]https://nipunbatra.github.io/blog/2017/recommend-keras.html

done on the training subset with a certain number of epochs (which need to be tuned) and validates on the test subset. This is done several times on different train/test subsets to ensure a good training of the system. The system is then fed with the whole dataset for the final prediction.

2) Collaborative Filtering with Neural Network: This model uses an neural network to do collaborative filtering (pretty much the same thing as the first model). To do so, we took inspiration of the keras_recommender library[5] based itself on keras. We took the same model as the *Collaborative-FilteringV1* presented on that git repository, and we added a dropout layer to prevent overfitting (see fig. 2).

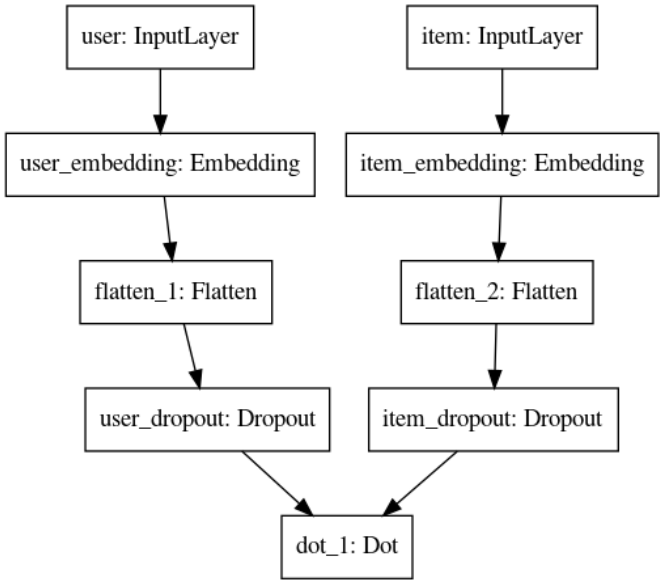

Fig. 2. Model for Neural Network Collaborative Filtering

Model fitting is done with the whole dataset, with a validation split of 20%.

III. Results

A. Cosine Similarities

After some very long computation ($\approx 16hours$), the result and the loss were not as good as expected. This method does not seem to be adapted to make prediction on a whole dataset, but only on one particular user/item entry.

B. Matrix Factorization

To find good result in our matrix factorization models, we started by using the splitting data method. To split the data, we looked at how many times a user had voted for movies and how many movies had votes. Then we selected all movies and users with at least $min\_num\_ratings$. To have a repartition of around 90% training set and 10% in our testing set, we chose $min\_num\_ratings$ = 10. This splitting method helps us to find a better RMSE when we search for our hyperparameters. To find them, we simply compute the RMSE for a lot of different hyperparameters and took the smallest RMSE.

1) Hyperparameters for Matrix Factorization using SGD: The parameters to find are γ, λ_p, λ_q, the number of latent features K and the number of iterations. We used the splitting method to find the parameters which gave us the smallest RMSE.

$$\gamma = 0.008$$

$$\lambda_p = 0.001$$

$$\lambda_q = 0.001$$

$$K = 20$$

$$iterations = 30$$

The RMSE obtained on CrowdAI is 1.044 (submission 24524 by userj) which ranks us at the 78th place, at 3:30 pm on 20th of December. To see the RMSE graph, see fig.3.

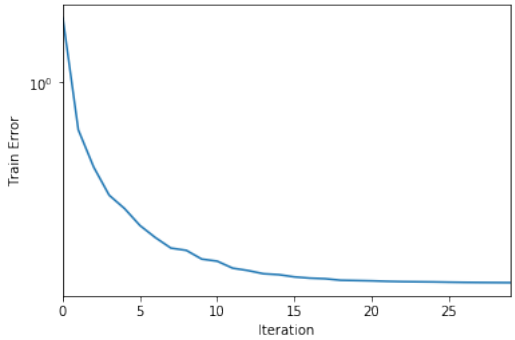

Fig. 3. Training error for the best hyperparameters for SGD model

2) Hyperparameters for Matrix Factorization using ALS:

$$\lambda_p = 0.008$$

$$\lambda_q = 0.008$$

$$K = 20$$

$$stop\_criterion = 1 \cdot 10^{-5}$$

The RMSE obtained on CrowdAI is 1.025 (submission 24642 by jvonfelte) which ranks us at the 40th place, at 3:30 pm on 20th of December. To see the RMSE graph, see fig.4.

C. Neural Network

The results of the neural network models were not as good as expected. Because those models were introduced late in this project, a lack of time prevented us to do some good parameter tuning.

[5]https://github.com/chen0040/keras-recommender

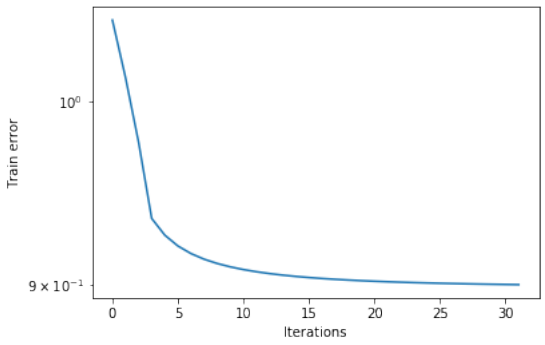

Fig. 4. Training error for the best hyperparameters for ALS model

1) Matrix Factorization: The parameters to tune for this model were the number of latent factors for users and movies (n_latent_factors_user and n_latent_factors_movie respectively in the code), the density of the Dense layers, the dropout factor and the number of epochs for the training. The best result we obtained where with the parameters given in the article mentioned above, i.e.

$$n\_latent\_factors\_user = 5$$
$$n\_latent\_factors\_item = 8$$
$$dropout = 0.2$$
$$epochs = 100$$
$$density \in [200, 100, 50, 20]$$

The history of the loss during the training is given by fig.5. As we can see, lots of epoch does not necessarily mean a better loss.

We obtained a loss of 0.7634983290349866 using MAE as a cost function, which lead to a score on CrowdAI of 1.072.

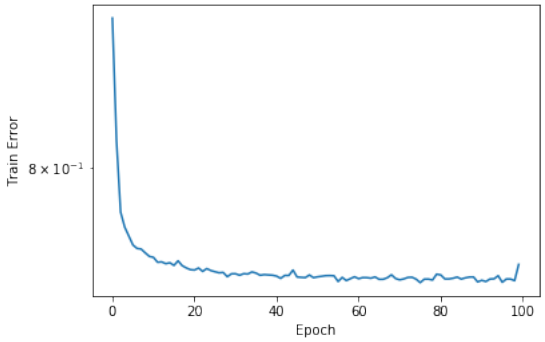

Fig. 5. History of loss during training - Factorization

2) Collaborative Filtering: For this model, only the dropout factor, the number of epochs and the output dimension of the embedding layers were tunable. As before, the best results were given by the initial values of the original code (c.f. the git repository of that library), which are:

$$epochs = 20$$
$$dropout = 0.2$$

in addition with a non-negativity constraint to the *Embedding* layers.

The history of the loss during training is given by fig.6.

Even though we get a loss of 0.712809626085298, the score we get on CrowdAI (1.095) is quite bad (submission 24882 by userj)

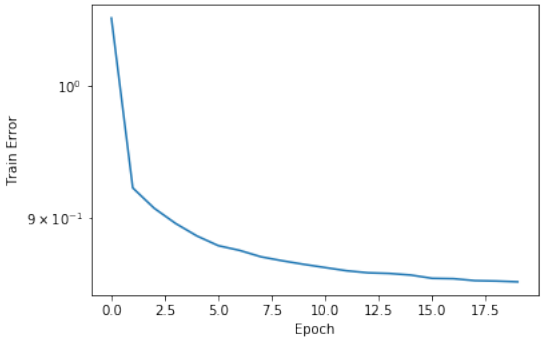

Fig. 6. History of loss during training - Collaborative Filtering

IV. Discussion

We tried different models and found that the best result was obtained using matrix factorization with ALS. Cosine similarities did not give decent results and took a lot of time to compute the values. Matrix factorization is better fitted to this type of problem, as the entries (user ratings for a movies) are easily represented as a matrix. Since running the algorithms takes a lot of time, we did not try all possible combinations of parameters, but for some arbitrarily chosen ones. We could also further improve our hyperparameters by using cross-validation with more splits.

Additionally, for neural network models, we were surprised to discover that a high number of epochs does not lead to a better loss. As an example, we tried to run the Matrix Factorization with Neural Network with both latent factors $= 32$ for 100 epochs. The history shows a quite chaotic loss that is not going better as the time goes (see fig.7).

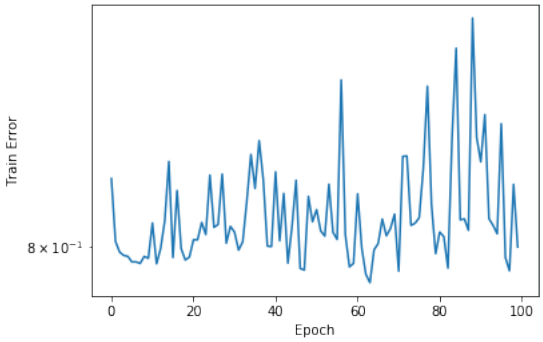

Fig. 7. Neural Network Matrix Factorization, 32 latent factors, 100 epochs

Hence, one thing we learned is that lots of epochs is not necessarily a good thing and it needs to be tuned with as much carefulness as other parameters.

V. Summary

With this recommender system challenge, we could understand how to conduct a machine learning project from the dataset to the predictions. We could see different methods as ALS, SGD, cosine and neural networks and their different results. We saw that cosine can be a really good method to recommend one film to a specific user but is not working well if we need to find every recommendations. ALS gave us the best result with a 1.025 RMSE which makes us in the top 40 in this challenge.