Semester project

Recommender Systems: Neural Collaborative Filtering vs. Matrix Factorization

Author: Florent DE BORTOLI

February 11, 2021

EURECOM
Sophia Antipolis

Contents

Introduction

Over the last few years, in the era of information explosion, recommender systems play a pivotal role in alleviating information overload, having been widely adopted by many online services, including E-commerce, online news and social media sites, with a lot of applications in various sectors. For example, Web giants such as Amazon use them to recommend some items to their customers who are likely to buy. On-demand and streaming platforms such as Netflix and Spotify suggest to their users some multimedia contents they would like to watch or listen. Airlines and rail companies benefit from recommender systems to propose some destinations to their customers who are likely to choose for their next trip. Apple and Google use them also to recommend applications on their stores to their users according to those they have already downloaded in the past, etc... There are huge economic concerns behind recommender systems, because they might increase significantly the turnover of companies using them. Indeed, personalizing suggestions on a service increases a lot the probability of purchase.

Recommender systems are currently the subject of many research works. They were quite simple several years ago, but now specialists benefit from the last breakthroughs in Artificial Intelligence and Deep Learning. Indeed, neural networks theory offers more accurate models by learning interactions between users and items. In this report, users and items are generic words that can be specified in the context of any recommender system. Thereby, methods and concepts may be practical to a lot of services. For example, items might denote goods on a e-commerce website, movies on Netflix or trip destinations for an airline.

In this document, we will present the cutting-edge methods currently used in the industry, that is to say the Matrix Factorization (MF), the Generalized Matrix Factorization (GMF), the Multilayer Perceptron (MLP) and finally the combination of these two previous ones which is called Neural Matrix Factorization (NeuMF). Models have been implemented in Python with Keras, which is the Deep Learning framework developped by Google. These models will be tested on two popular datasets: MovieLens and Pinterest.

This project is inspired by the work presented by the papers [1] and [2]. The first paper introduces the concept of learned similarity with deep neural networks (MLP and NeuMF). The authors firstly recall the usual methods of recommendation: the MF and the GMF. Then, they conduct some experiments on the datasets MovieLens and Pinterest to demonstrate the effectiveness of such new methods over usual models. The second paper revisits the experiments conducted in NCF paper and demonstrates that finally, with a proper hyper-parameter selection, a dot product outperforms any learned similarity.

The purpose of this project is to reproduce the experiments performed in these two papers and to conclude on the benefits of using learned similarities. You can find all of the code and experiments presented in this document in my GitHub repository by clicking on the link below. [1] In this repository, there are the data we worked with and a jupyter notebook including everything.

[1]**https://github.com/fdb78/NCF_recommender_systems**

1 Formalism of recommender systems

Firstly, let's formalize the overall working of recommender systems. There exists two main approaches to make recommendation. Content-based filtering uses item features to recommend other items similar to what the user likes, based on their previous actions or explicit feedback. This approach has become little by little outdated and it has been replaced by the collaborative filtering. This second approach leverages the similarities between users and items simultaneously to make recommendations.

Let's consider a set of M users and a set of N items denoted respectively by $U = \{u = 1, ..., M\}$ and $I = \{i = 1, ..., N\}$. A usual recommendation dataset is composed of a list of many interactions, each one is represented by a triple (u, i, y), where u is the user ID, i is the item ID. y denotes the score of the interaction, it is either a binary value for an implicit feedback, either a integer score between 0 and 5 for example that measures how much the user u like or dislike the item i in the case of explicit feedback.

There exists two main types of recommendation problem:

- **Explicit feedback** consists in collecting ratings and reviews by asking in an explicit way the users to mark a content. For example in e-commerce website you may be asked to rate the articles that you bought or the movies you watched on a streaming plateform. In this case, the negative feedbacks are readily available in the dataset because the score of interaction y quantifies how much a user like or dislike an item.

- **Implicit feedback** consists in analysing the behaviour of users without their awareness through clicks, purchases, etc... It offers the advantage to be easier to collect but the dataset does not contain any negative feedback, which could be problematic to offer good performance.

For this project we focus on implicit feedback because it is easier to deal with, so $y = 1$ means that the interaction has been observed between u and i and $y = 0$ means that this entry has not been reported by the system of collection. It is notable that $y = 1$ just means that the user u has interacted with the item i, but does not mean that he likes i. In the same way, $y = 0$ does not mean that the user u dislike i but just mean that he did not interacted with it. So as you read before, the implicit feedback brings about a lack of negative feedback but you will see that we are still able to get interesting results.

The ultimate goal of a recommender system is to estimate the score $\hat{y}$ of every unobserved entry. The way of estimation and its performance is the object of many researchers. Then, for each user the system aims to rank all the items from the highest score to the lowest in order to maximize its satisfaction and to suggest the items that he is more likely to like.

This problem is considered as a function approximation problem, the function we want to fit takes the pair (u, i) as an argument and returns the score of interaction y between u and i. This function is called the interaction function, or equivalently the similarity function, because it represents the interaction between users and items. Thus, we can solve this problem using the Machine Learning paradigm that involve fitting a model on data. Formally speaking, we want to estimate the score of interaction :

$$\hat{y}_{u,i} = f(u, i|\theta)$$

where $\hat{y}_{u,i}$ denotes the estimated score of interaction between user u and item i. In this case of implicit feedback, this is also the probability that u would interact with i because the score belongs to $[0, 1]$. θ denotes the model parameters which are computed by a usual optimization in Machine Learning, which consists in minimizing a loss function. This loss function measures the error between the output of the model and the target we want to approach. The loss function used here is the binary cross-entropy which is perfectly appropriate for this problem. The minimization is performed by the Gradient Descent algorithm and its famous variants Mini-Batch Gradient Descent and Stochastic Gradient Descent (SGD).

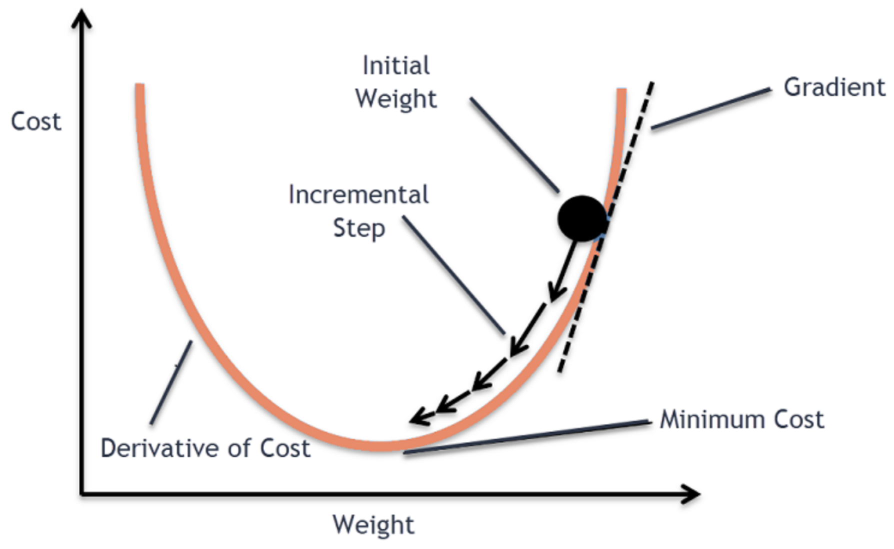


Figure 1: Optimization of a loss function using Gradient Descent

In recommender systems, we usually work with **embedding** based models. But let's explain what is an embedding. The triples of the dataset that we introduced before are actually used to build a big matrix called the user-item interaction matrix. In this matrix, each row is a user and each column is an item, and the matrix is filled with the scores of interaction. Then, the dimensions of this matrix is $M \times N$. This matrix is very sparse because each user has interacted with just a few items among the lot of existing items. It is even sparse at 99% for most of datasets because users and items are represented by one-hot encoded vectors. Sending such a vector as input to a model would be inefficient because the input dimension would be too high and too sparse. Thus, a dimension reduction technique maps these huge and sparse one-hot encoded vectors of users and items to embedding vectors in a common latent space $E = \mathbb{R}^d$. Typically, the embedding space is low-dimensional (that is, d is much smaller N and M), and captures some latent structures. It projects a discrete set to a vector space, so embedding vectors are no longer binary and they are filled with real number. This is very much more computationally efficient when using huge and sparse datasets. The outputs of an embedding layer are also called the latent vectors, and its elements are the latent factors.

The embedding layer is in fact as a simple perceptron with one single layer and no activation function, and its weights are trained by optimization. This approach comes from the Natural Language Processing theory and Word2Vec algorithms where each word is one-hot encoded among all the words of the dictionary, which is quite similar to the sparsity of recommendation datasets. In recommender systems, the concatenation of a user embedding with an item embedding is sent as input to a model that returns a single score as output that indicates the likeness between the user and the item.

2 Popular Models Used in Recommender Systems

2.1 Matrix Factorization: Dot Product

Let's introduce the first model, which is very simple: the Matrix Factorization (MF). In the MF, the interaction function f is a dot product. The interaction score between a user u and an item i is equal to the dot product between their embedding vectors. This method is named Matrix Factorization because it is performed by factorizing the big and sparse user-item interaction matrix into two smaller matrices containing the latent factors (*i.e.* the embedding outputs). One of these matrices contains the user embeddings and the other contains the item embeddings. The factorization is performed by optimization. The loss function that has to be minimized measures the divergence between the observed entries and the dot product between the corresponding user embedding and the item embedding.

Mathematically speaking, the factorization of the user-item interaction matrix boils down to exactly the same thing that computing embeddings with a simple perceptron and then to apply the dot product between them. Indeed, every element of a matrix product is equal to the dot product between the line of the first matrix and the columns of the second one, thus between the user embedding and the item embedding. This picture just below summarizes in a clear way how the MF works. In this case the items are the movies watched by users of a streaming plateform like Netflix.

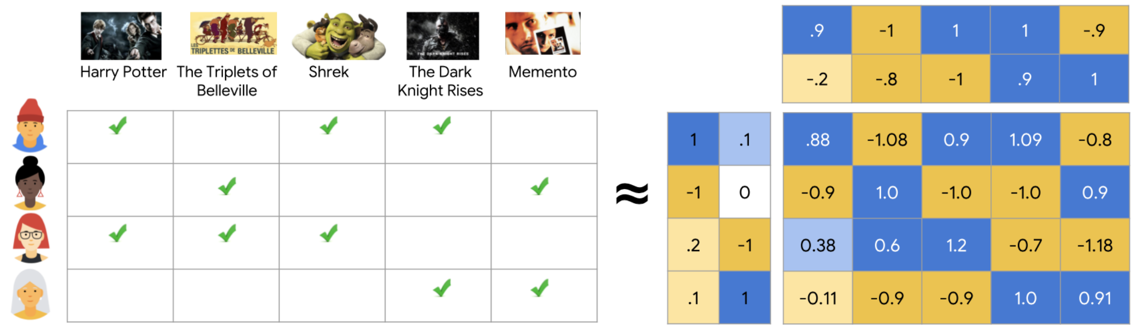


Figure 2: Matrix Factorization

Let p_u and q_i denote the latent vectors for user u and item i respectively, then MF estimates the interaction score $\hat{y_{u,i}}$ as the inner product between p_u and q_i:

$$\hat{y_{u,i}} = f(u, i|\theta) = \langle p_u, q_i \rangle = p_u^T q_i = \sum_{k=1}^{K} p_{uk} q_{ik}$$

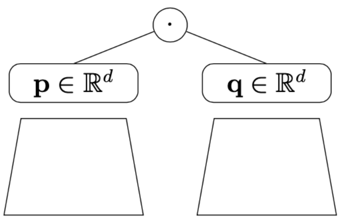


Figure 3: Evaluation of the score using MF

where K denotes the dimension of the latent space and θ denotes the parameters of the model, *i.e.* the weights of the embedding layer. Thus, MF can be deemed as a linear model of latent factors. The objective function to minimize is the squared distance, equal to the sum of squared errors over all pairs of observed entries. In this objective function, it is worth noting that we only sum over observed pairs (u, i).

$$\mathscr{L} = \sum_{(i,j) \in obs} (y_{i,j} - \hat{y_{i,j}})^2 = \sum_{(i,j) \in obs} (y_{i,j} - p_u^T q_i)^2$$

However, the inner product function can limit the expressiveness of MF. Indeed, using such a simple and fixed operation between users and items embeddings might offer poor performances to estimate the interaction function which is much more complex in reality, and it is hard to approximate efficiently with such a linear model. Solving this issue without changing the model structure would require to use a large number of latent factors, but it may hurt the generalization of the model and lead to overfitting. The problem comes from the linearity, which is inherent to MF.

2.2 Generalized Matrix Factorization

The Generalized Matrix Factorization (GMF) may improve a bit the previous model by weighting the dot product, where the weights are learnt from data by optimization. Another improvement is to pass the result through an non-linear activation function. This enables to add some non-linearity in the model and to make it a bit more complex, so this model is supposed to offer better performance than simple MF. The activation function is the sigmoid function:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Let's keep the same notations we used previously for MF, the GMF uses the following interaction function f to estimate the score between p_u and q_i:

$$\hat{y_{u,i}} = f(u, i|\theta) = \sigma(w^T(p_u \odot q_i)) = \sigma\left(\sum_{k=1}^{K} w_k p_{uk} q_{ik}\right)$$

In this formula, the operator $\odot$ represents the element-wise multiplication. Unlike the dot product which returns a scalar, this operation returns a vector, that is composed by the product of each elements two by two. We will explain later that the GMF is actually just a particular case of the next model MLP.

2.3 Learned similarity: Multilayer Perceptron

Now let's talk about the core model introduced recently in the research by the papers : the Multilayer Perceptron (MLP). The neural network has been proven to be capable of approximating any continuous function, and more recently deep neural networks have been found to be effective in several domains, ranging from computer vision, speech recognition, to text processing. This model is very common in Deep Learning, this is the first neural network, made up of several stacked layers of neurons with activation functions after each layer. For recommender systems, we send as input the concatenation of the embeddings of a user and the embedding of an item as we did for the two previous models. The output is composed by one single unit that merges the results of all previous layers and applies a last activation function to yield the score of the interaction between 0 and 1. The training of the network consists in forwarding the samples of the dataset through the network. Before the training, the weights are randomly initialized following a standard normal distribution. Then, the output score is compared with the target score through the loss function that we want to minimize. Finally the weights fit to the data thanks to the back-propagation of the error through the network. All these steps constitute what we called an epoch, and the number of epochs is an important hyper-parameter that we can set empirically by analyzing the learning curves. The learning curves plot the losses (training and testing loss) and other different metrics specific to the problem with the epoch number. By inspecting these curves, we can detect when the model is correctly fit and we can prevent overfitting.

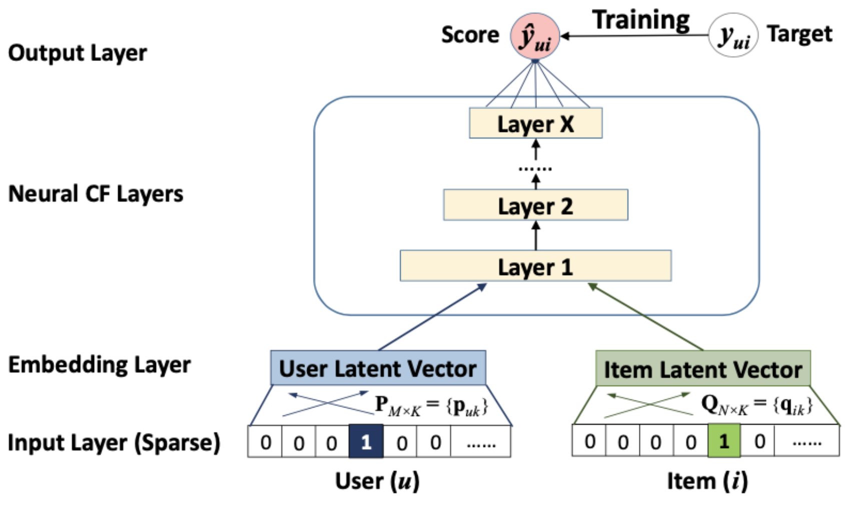


Figure 4: Multilayer Perceptron

Regarding the architecture of the network, a common solution is to follow a tower pattern, where the bottom layer is the widest and each successive layer has a smaller number of neurons (as in the previous figure). Indeed, using a small number of hidden units for higher layers allow them to learn more abstractive features of data. For activation functions we have the choice between using sigmoid σ, hyperbolic tangent (tanh) and Rectified Linear Unit also known as ReLU. Some empirical results from the papers show that ReLU yield better performance than the two other ones so we will use it for the experiments.

As we said previously, the MLP can be deemed as an extra generalization of the GMF, because the GMF is just a particular case of MLP where the input is directly connected to the output unit without any hidden layer, and the activation function of the input is the identity function. In this case, the weighted dot product in GMF model is exactly recovered. If we add a uniform constraint on the weights to be uniformly equal to one and we take the identity function as the last activation, we even recover the simple dot product from the first model MF.

2.4 Neural Matrix Factorization

In this part we are going to introduce the final model, this is the most complex. It is supposed to offer the best performance among all previous models. It is called Neural Matrix Factorization (NeuMF). This model is the combination of the two previous models (GMF and MLP), so that they can mutually reinforce each other to better model the complex user-item interaction function. This model leverages the linearity of the GMF and the non-linearities from activation functions in neural networks. To provide more flexibility to the fused model, we allow GMF and MLP to learn separate embeddings, and combine the two models by concatenating their last hidden layer. You can find in the figure below a schema of the NeuMF model.

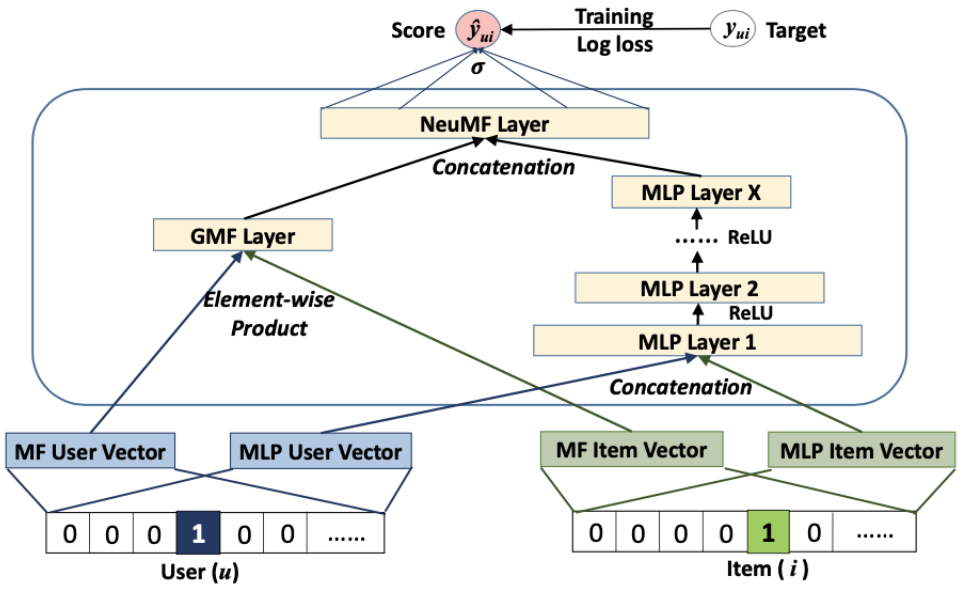


Figure 5: Neural Matrix Factorization

The authors of the first paper [1] propose an additional improvement which consist in pre-training separately the GMF part and the MLP part with random initialization for the weights, and then using these weights as the initialization for the corresponding parts of NeuMF's parameters. However, we will not deal with this tweak in the experiments but it could be included in future work.

3 Implementation of models with Keras and TensorFlow

Now you should be familiar with the state-of-the-art theory and methods used in most of recommender systems, it is time to think about implementation and application. In this section you will find how I have implemented the four models I described in the previous part. I recall that the models are the MF, the GMF, the MLP and the NeuMF. Even if you can find my code in my GitHub repository, I included some parts of my code in this report. Of course, I used the language Python with the usual Data Science libraries such as Numpy, Pandas and Matplotlib to import and process the dataset. For the model part I used Keras with a TensorFlow back-end, this configuration is very convenient for Data Science because Keras is a high-level API so it is quite straightforward to build from simple models to complex neural networks. TensorFlow is a powerful tool designed by Google and one of the most popular with PyTorch in the field of AI and Deep Learning.

I developed one function by model to generate a new model with new weights. All these functions require at least the number of users and the number of items in the dataset, and also the number of predictive factors (*i.e.* the dimension of the last hidden layer before the ultimate score neuron) as parameters. The two first parameters are required in order to set the embedding layers at the beginning of the model. The number of predictive factors determines the model capability and we will observe the influence of this parameter in the next section (Experiment 3). In each model implementation, the variable d contains the dimension of the embedding dimensions. Indeed, we must provide the input and output dimension to build an embedding layer. This variable `d` is computed according to the number of predictive factors which is specified in argument.

We perform the optimization with the Adam optimizer which is very usual and offers a fast convergence to the minimum of the loss function. The best loss function to optimize here is the binary cross-entropy also named log loss, this is the one that is the most appropriate for the recommendation problem. This function measures the error of prediction.

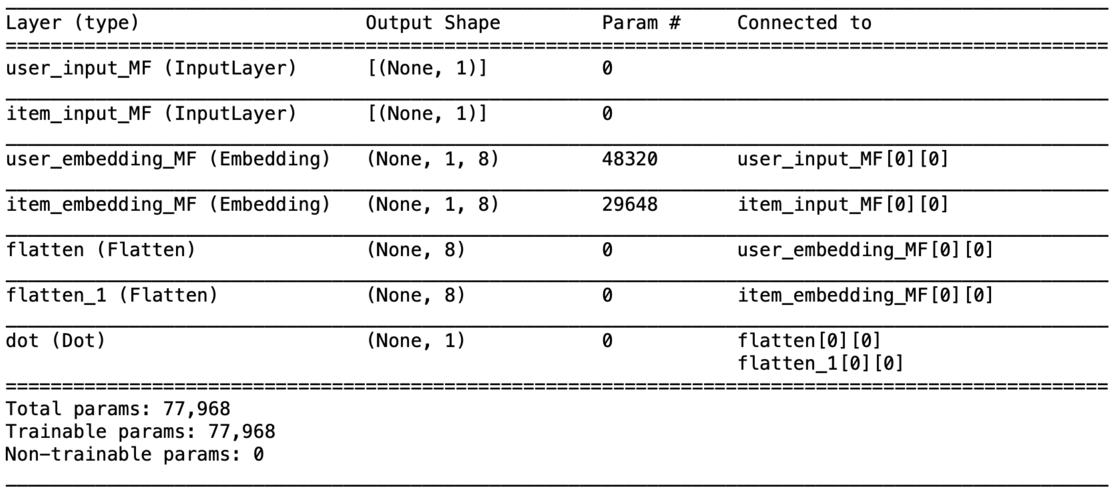
3.1 Matrix Factorization

Let's start with the simplest model : the MF. As we introduced it in the first section, this model evaluates the score of interaction between a user and an item by computing the dot product of their embeddings. So in the following implementation, you can find the two embedding layers taking as input the input of the model and yielding a smaller vector. In this model, the predictive factors correspond exactly to the element-wise multiplication of the two embeddings. The dimension of the element-wise multiplication is the same as the dimension of the embeddings themselves. Thus, the output dimension of embeddings must be exactly the number of predictive factors. We set the default value as 8. The embeddings are flattened to fit the input of the next layers. Then, the flattened embeddings are sent to a dot layer which compute the score. You can find the code of MF in the figure just below.

```python
def get_model_MF(num_users, num_items, factors=8):
    d = factors

    user_input_MF = Input(shape=(1,), dtype='int32', name='user_input_MLP')
    item_input_MF = Input(shape=(1,), dtype='int32', name='item_input_MLP')

    user_embedding_MF = Embedding(input_dim=num_users, output_dim=d, name='user_embedding_MLP')(user_input_MF)
    item_embedding_MF = Embedding(input_dim=num_items, output_dim=d, name='item_embedding_MLP')(item_input_MF)

    user_latent_MF = Flatten()(user_embedding_MF)
    item_latent_MF = Flatten()(item_embedding_MF)

    dot = Dot(axes=1)([user_latent_MF, item_latent_MF])

    MF = Model(inputs=[user_input_MF, item_input_MF], outputs=dot)
    MF.compile(optimizer=tf.train.AdamOptimizer(), loss=binary_crossentropy)
    return MF
```

Listing 1: Implementation of MF

You can also find below the summary of the generated model. As you can observe, there are around 77,968 weights to train. All these weights come from the embedding layers, because the dot product is not trainable., it is a non-parametric layer.



```
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
user_input_MF (InputLayer)      [(None, 1)]          0
__________________________________________________________________________________________________
item_input_MF (InputLayer)      [(None, 1)]          0
__________________________________________________________________________________________________
user_embedding_MF (Embedding)   (None, 1, 8)         48320       user_input_MF[0][0]
__________________________________________________________________________________________________
item_embedding_MF (Embedding)   (None, 1, 8)         29648       item_input_MF[0][0]
__________________________________________________________________________________________________
flatten (Flatten)               (None, 8)            0           user_embedding_MF[0][0]
__________________________________________________________________________________________________
flatten_1 (Flatten)             (None, 8)            0           item_embedding_MF[0][0]
__________________________________________________________________________________________________
dot (Dot)                       (None, 1)            0           flatten[0][0]
                                                                 flatten_1[0][0]
==================================================================================================
Total params: 77,968
Trainable params: 77,968
Non-trainable params: 0
__________________________________________________________________________________________________
```

Figure 6: Summary of model MF

3.2 Generalized Matrix Factorization

Now, let's present the GMF code. It looks quite similar with the code of MF, indeed you can find the two same embedding layers at the beginning of the model. The difference is that we perform an element-wise multiplication of the two embeddings and we send the resulting vector to a dense layer. The element-wise multiplication is performed by the layer `Multiply()` from Keras. The last dense layer is a simple perceptron, it allows to weights the elements in the sum unlike the dot product. It uses the sigmoid as activation function that adds non-linearities compared to MF.

Here the predictive factors still correspond exactly to the element-wise multiplication so I have not changed anything regarding the dimension of embeddings.

```python
def get_model_GMF(num_users, num_items, factors=8):
    d=factors
    user_input_GMF = Input(shape=(1,), dtype='int32', name='user_input_GMF')
    item_input_GMF = Input(shape=(1,), dtype='int32', name='item_input_GMF')

    user_embedding_GMF = Embedding(input_dim=num_users, output_dim=d, name='user_embedding_GMF')
    item_embedding_GMF = Embedding(input_dim=num_items, output_dim=d, name='item_embedding_GMF')

    user_latent_GMF = Flatten()(user_embedding_GMF(user_input_GMF))
    item_latent_GMF = Flatten()(item_embedding_GMF(item_input_GMF))

    mul = Multiply()([user_latent_GMF, item_latent_GMF]) # len = factors

    prediction_GMF = Dense(units=1, activation='sigmoid', name='prediction')(mul)

    GMF = Model(inputs=[user_input_GMF, item_input_GMF], outputs=prediction_GMF)
    GMF.compile(optimizer=tf.train.AdamOptimizer(), loss=binary_crossentropy)
    return GMF
```

Listing 2: Implementation of GMF

As previously you can find below the summary generated by Keras on this model. Here we set the number of predictive factors at 8 like we do for every model in order to get comparable results without tuning this parameter. We observe that the number of trainable parameters is almost the same as the one for MF, which is logical because the only additional trainable parameters come from the weights in the dot product, their number is equal to the number of predictive factors. But the number of parameters coming from the embedding layers remain exactly the same.

```
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
user_input_GMF (InputLayer)     [(None, 1)]          0
__________________________________________________________________________________________________
item_input_GMF (InputLayer)     [(None, 1)]          0
__________________________________________________________________________________________________
user_embedding_GMF (Embedding)  (None, 1, 8)         48320       user_input_GMF[0][0]
__________________________________________________________________________________________________
item_embedding_GMF (Embedding)  (None, 1, 8)         29648       item_input_GMF[0][0]
__________________________________________________________________________________________________
flatten_6 (Flatten)             (None, 8)            0           user_embedding_GMF[0][0]
__________________________________________________________________________________________________
flatten_7 (Flatten)             (None, 8)            0           item_embedding_GMF[0][0]
__________________________________________________________________________________________________
multiply_1 (Multiply)           (None, 8)            0           flatten_6[0][0]
                                                                 flatten_7[0][0]
__________________________________________________________________________________________________
prediction (Dense)              (None, 1)            9           multiply_1[0][0]
==================================================================================================
Total params: 77,977
Trainable params: 77,977
Non-trainable params: 0
__________________________________________________________________________________________________
```

Figure 7: Summary of model GMF

3.3 Multilayer Perceptron

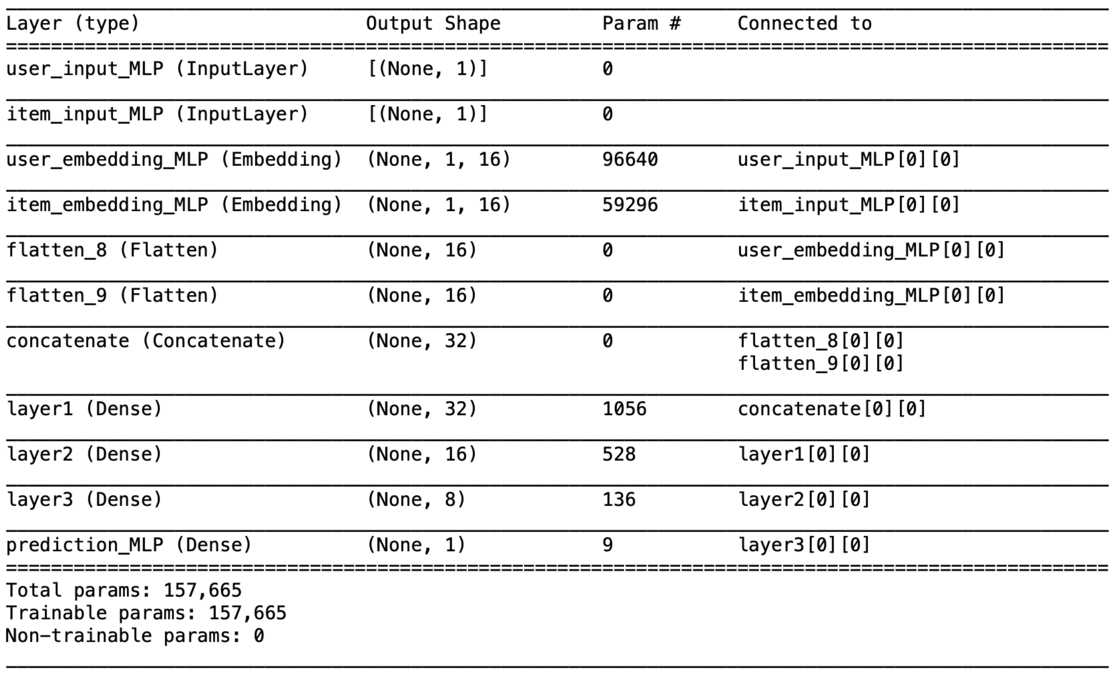
The model MLP takes the architecture of the neural network as an additional parameter: the number of layers. We set its default value to 3 as it is in the paper [1] but we will study later their influence within the experiment section. We keep the tower pattern that we explained in the previous section, where each layer has a number of units equal to a power of 2. Each layer has twice the number of neurons its next layer has, thus the entire structure of a model is determined by these two hyper-parameters. We can build the network and find the embedding output dimension `d` by doing a quick reasoning that we will not explain in details here.

The beginning of the model is still the same as the other ones, with two embedding layers (one for the user and one for the item) and then they are flattened and concatenated by the already implemented layer from Keras. The concatenation is then forwarded through the network until the last score unit.

```python
def get_model_MLP(num_users, num_items, num_layers=3, factors=8):
    if num_layers==0:
        d = int(factors/2)
    else:
        d = (2**(num_layers-2))*factors

    user_input_MLP = Input(shape=(1,), dtype='int32', name='user_input_MLP')
    item_input_MLP = Input(shape=(1,), dtype='int32', name='item_input_MLP')

    user_embedding_MLP = Embedding(input_dim=num_users, output_dim=d, name='user_embedding_MLP')(user_input_MLP)
    item_embedding_MLP = Embedding(input_dim=num_items, output_dim=d, name='item_embedding_MLP')(item_input_MLP)

    user_latent_MLP = Flatten()(user_embedding_MLP)
    item_latent_MLP = Flatten()(item_embedding_MLP)

    concatenation = Concatenate()([user_latent_MLP, item_latent_MLP])
    output = concatenation
    layer_name = 0
    for i in range(num_layers-1,-1,-1):
        layer = Dense(units=(2**i)*factors, activation='relu', name='layer%d' %(layer_name+1))
        output = layer(output)
        layer_name += 1
    prediction_MLP = Dense(units=1, activation='sigmoid', name='prediction_MLP')(output)
    MLP = Model(inputs=[user_input_MLP, item_input_MLP], outputs=prediction_MLP)
    MLP.compile(optimizer=tf.train.AdamOptimizer(), loss=binary_crossentropy)
    return MLP
```

Listing 3: Implementation of MLP

In the MLP model, the number of trainable parameters is much higher than the previous models (for the same number of predictive factors of course) because the stacking of several hidden layers multiply the number of weights.



```
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
user_input_MLP (InputLayer)     [(None, 1)]          0
__________________________________________________________________________________________________
item_input_MLP (InputLayer)     [(None, 1)]          0
__________________________________________________________________________________________________
user_embedding_MLP (Embedding)  (None, 1, 16)        96640       user_input_MLP[0][0]
__________________________________________________________________________________________________
item_embedding_MLP (Embedding)  (None, 1, 16)        59296       item_input_MLP[0][0]
__________________________________________________________________________________________________
flatten_8 (Flatten)             (None, 16)           0           user_embedding_MLP[0][0]
__________________________________________________________________________________________________
flatten_9 (Flatten)             (None, 16)           0           item_embedding_MLP[0][0]
__________________________________________________________________________________________________
concatenate (Concatenate)       (None, 32)           0           flatten_8[0][0]
                                                                 flatten_9[0][0]
__________________________________________________________________________________________________
layer1 (Dense)                  (None, 32)           1056        concatenate[0][0]
__________________________________________________________________________________________________
layer2 (Dense)                  (None, 16)           528         layer1[0][0]
__________________________________________________________________________________________________
layer3 (Dense)                  (None, 8)            136         layer2[0][0]
__________________________________________________________________________________________________
prediction_MLP (Dense)          (None, 1)            9           layer3[0][0]
==================================================================================================
Total params: 157,665
Trainable params: 157,665
Non-trainable params: 0
__________________________________________________________________________________________________
```

Figure 8: Summary of model MLP

3.4 Neural Matrix Factorization

Finally, let's present the code of the NeuMF model, which is the fuse of models GMF and MLP. There is a GMF part and an MLP part that are exactly the same as in the two previous models. You can notice that each part uses its own embedding layer in order to improve performance because there are less constraints on each part, so there are 4 embedding layers in total. The parts even do not use the same embedding dimensions. The output vectors coming from each part are finally concatenated before being sent to the final score unit. The arguments `num_layers_MLP_part` and `factors` specify entirely the architecture of the whole model, as it is in the code of model MLP thanks to the tower pattern. I set the same default value for this argument as in the MLP model.

In this model the number of predictive factors is a bit more complicated to understand, because there are some predictive factors coming from the GMF part and other predictive factors coming from the MLP part. A tweak proposed by the paper [1] would set the trade-off α between the two parts but I have not implemented this tweak in this project. If you have a closer look at the code, you can understand that the number of predictive factors is equal to the sum of the dimension of the element-wise multiplication from the GMF part and the dimension of the output of MLP part. As we did for the previous model MLP, a quick reasoning on the architecture of the network can lead to the number of units per layer and even until the embeddings output dimensions for each part.

```python
def get_model_NeuMF(num_users, num_items, num_layers_MLP_part=3, factors=8):
    assert (factors%2)==0
    if num_layers_MLP_part==0:
        d_MLP = int(factors/4)
    else:
        d_MLP = (2**(num_layers_MLP_part-3))*factors

    user_input = Input(shape=(1,), dtype='int32', name='user_input_NeuMF')
    item_input = Input(shape=(1,), dtype='int32', name='item_input_NeuMF')

    ## MLP part
    user_embedding_MLP = Embedding(input_dim=num_users, output_dim=d_MLP, name='user_embedding_MLP')(user_input)
    item_embedding_MLP = Embedding(input_dim=num_items, output_dim=d_MLP, name='item_embedding_MLP')(item_input)

    user_latent_MLP = Flatten()(user_embedding_MLP)
    item_latent_MLP = Flatten()(item_embedding_MLP)

    concatenation_embeddings = Concatenate()([user_latent_MLP, item_latent_MLP])

    output_MLP = concatenation_embeddings
    layer_name = 0
    for i in range(num_layers_MLP_part-2,-2,-1):
        layer = Dense(units=(2**i)*factors, activation='relu', name='layer%d' %(layer_name+1))
        output_MLP = layer(output_MLP)
        layer_name += 1

    d_GMF = int(factors/2)
    ## GMF part
    user_embedding_GMF = Embedding(input_dim=num_users, output_dim=d_GMF, name='user_embedding_GMF')
    item_embedding_GMF = Embedding(input_dim=num_items, output_dim=d_GMF, name='item_embedding_GMF')

    user_latent_GMF = Flatten()(user_embedding_GMF(user_input))
    item_latent_GMF = Flatten()(item_embedding_GMF(item_input))

    mul = Multiply()([user_latent_GMF, item_latent_GMF])

    concatenation_of_models = Concatenate(name='final_concatenation')([mul, output_MLP]) # len = factors
    prediction_NeuMF = Dense(units=1, activation='sigmoid', name='prediction')(concatenation_of_models)

    NeuMF = Model(inputs=[user_input, item_input], outputs=prediction_NeuMF)
    NeuMF.compile(optimizer=tf.train.AdamOptimizer(), loss=binary_crossentropy)
    return NeuMF
```

Listing 4: Implementation of NeuMF

As usually you can find the summary of the model NeuMF in the figure below. You can observe that for the same number of predictive factors (8 in these figures) there are less trainable parameters in NeuMF than in MLP. This comes from that half of the predictive factors come from the GMF part, which has very few parameters compared to MLP.

```
____________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
====================================================================================================
user_input_NeuMF (InputLayer)   [(None, 1)]          0
____________________________________________________________________________________________________
item_input_NeuMF (InputLayer)   [(None, 1)]          0
____________________________________________________________________________________________________
user_embedding_MLP (Embedding)  (None, 1, 8)         48320       user_input_NeuMF[0][0]
____________________________________________________________________________________________________
item_embedding_MLP (Embedding)  (None, 1, 8)         29648       item_input_NeuMF[0][0]
____________________________________________________________________________________________________
flatten_14 (Flatten)            (None, 8)            0           user_embedding_MLP[0][0]
____________________________________________________________________________________________________
flatten_15 (Flatten)            (None, 8)            0           item_embedding_MLP[0][0]
____________________________________________________________________________________________________
concatenate_3 (Concatenate)     (None, 16)           0           flatten_14[0][0]
                                                                 flatten_15[0][0]
____________________________________________________________________________________________________
user_embedding_GMF (Embedding)  (None, 1, 4)         24160       user_input_NeuMF[0][0]
____________________________________________________________________________________________________
item_embedding_GMF (Embedding)  (None, 1, 4)         14824       item_input_NeuMF[0][0]
____________________________________________________________________________________________________
layer1 (Dense)                  (None, 16)           272         concatenate_3[0][0]
____________________________________________________________________________________________________
flatten_16 (Flatten)            (None, 4)            0           user_embedding_GMF[0][0]
____________________________________________________________________________________________________
flatten_17 (Flatten)            (None, 4)            0           item_embedding_GMF[0][0]
____________________________________________________________________________________________________
layer2 (Dense)                  (None, 8)            136         layer1[0][0]
____________________________________________________________________________________________________
multiply_3 (Multiply)           (None, 4)            0           flatten_16[0][0]
                                                                 flatten_17[0][0]
____________________________________________________________________________________________________
layer3 (Dense)                  (None, 4)            36          layer2[0][0]
____________________________________________________________________________________________________
final_concatenation (Concatenat (None, 8)            0           multiply_3[0][0]
                                                                 layer3[0][0]
____________________________________________________________________________________________________
prediction (Dense)              (None, 1)            9           final_concatenation[0][0]
====================================================================================================
Total params: 117,405
Trainable params: 117,405
Non-trainable params: 0
____________________________________________________________________________________________________
```

Figure 9: Summary of model NeuMF

4 Datasets and environment setup for experiments

This section presents the datasets on which we will conduct some experiments later and the way I run the training and the evaluation of my models. The experiments have been conducted on the two famous datasets including the famous MovieLens and Pinterest, they are quite usual to test a model in the field of recommender systems. We are going to deal with the evaluation protocol and present several useful functions that have been run throughout the project. We also conducted some experiments on an industrial dataset called Airline Travel Dataset.

4.1 MovieLens and Pinterest datasets

MovieLens is a web-based recommender dataset that recommends movies for its user to watch, based on their film preferences using collaborative filtering of member's movie ratings and reviews. We used the version containing only one million ratings, and we selected the users having rated at least 20 movies. Originally it is an explicit feedback dataset because users are invited to rate the movies with their consent, but we transformed it into an implicit feedback by converting the interaction scores into binary values. Every observed entry score is set to 1, and unobserved entries scores at zero. We did this in order to make things simpler at the beginning, and to conduct the experiments in the same conditions as they did in the paper [1].

Pinterest is a social network focused on the sharing of pictures. In this dataset, each interaction denotes whether the user has pinned the image to its board. The dataset we used is only a subset of the whole dataset. As we did for MovieLens, we selected the users who have pinned at least 20 images to their board.

Dataset	**Interactions#**	**Users#**	**Items#**	**Sparsity**
MovieLens	1,000,209	3,706	6040	95,53%
Pinterest	1,500,809	9,916	55,187	99,73%

Table 1: Statistics of the datasets MovieLens and Pinterest

This table presents some statistics on dataset MovieLens and Pinterest. You can notice the very high sparsity of both datasets. It is logical because every user has only interacted with very few items among the huge number of items. This is characteristic of recommender systems, hence the purpose of the embedding layers to project data into a smaller space. For example on the e-commerce platform Amazon, you should understand that the number of items you are interested to is very small compared to the diversity of Amazon's catalog.

4.2 Processing of datasets

Each dataset is composed by two files : one containing the training set and another containing the test set. The first file is a very long list of triples in the form of (u, i, y) where u is a user, i is an item and y is the score of the interaction. As we explained before, y belongs to [1-5] for dataset MovieLens because the feedback is explicit but we will transform into 1, and y is already 1 for dataset Pinterest because the feedback is implicit. The first thing to do is to store all triples in a list (Numpy array).

The training file contains only positive interactions, which means that every interaction have been observed and has a score of 1. We cannot fit the models if only one class is represented in the dataset, so we need to get negative interactions with a score of 0. For that, we decide for each interaction to randomly select N items the user has not interacted with, in this way we generate many negative interactions in the train set and the dataset becomes bigger. N is a parameter that we can tune later to evaluate the models, it is called the sampling ratio. We are careful that there is no overlap between positive and negative interaction which would disrupt the model. The test file contains one long list per user, this list includes first the positive item that have been selected for the test set, and the 99 randomly selected items among these with he has not interacted. I had to perform myself the sampling of negative interactions.

4.3 Evaluation protocol

To evaluate performance of item recommendation, we adopt the **leave-one-out** evaluation which has been widely used. For each user, we held out its latest positive interaction as the test set and we keep the remaining data for training. This positive interaction is called the positive item. Since it is too time-consuming to rank all items for every user, we randomly select 99 items that have not been interacted with the user, so the test set contains 100 items to rank for each user. The test set was prepared beforehand for both datasets by the authors of the paper [1], so I had not to implement the random selection of the 99 items for each user to build the test set. Then we rank the 100 items by predicting scores with the model we want to evaluate, and we observe the position of the positive item in the ranklist. The ranklist denotes the list of items ordered by scores predicted by the model we want to evaluate.

The performance of a ranked list is judged by two metrics : the *Hit Ratio* (HR) and *Normalized Discounted Cumulative Gain* (NDCG). Intuitively, the HR measures whether the positive item belongs to the top-K in the list, where K is a parameter that we will tune later.

$$HR(r) = \begin{cases} 1 & \text{if } r \in [1; K] \\ 0 & \text{otherwise} \end{cases}$$

Unlike HR, NDCG takes into account the rank of the positive item to yield a score between 0 and 1. The better the rank is, the higher the score returned by NDCG is, whereas HR is just a binary value. To make the NDCG decreasing when the rank drops is to use a log at the denominator like in the following formula, where r denotes the rank of the positive item ($r >= 1$). With this formula, the NDCG is one is the rank r is equal to 1, which is the best case, and it is very low if r is big in the worst case. This metric is perfectly appropriate to evaluate a ranklist.

$$NDCG(r) = \begin{cases} \frac{log(2)}{log(1+r)} & \text{if } r \in [1; K] \\ 0 & \text{otherwise} \end{cases}$$

Then we compute these metrics for every user of the dataset and we report the average. You can have a look at the following code snippet showing the two functions we use to compute the metrics. These two functions `getHitRatio()` and `getNDCG()` take as argument the ranklist of the 100 items, the parameter K and the positive item (i.e. the one that we extracted from the dataset and which we know that the user has interacted with).

```python
def getHitRatio(ranklist, K, positive_item):
    if positive_item in ranklist[:K]:
        return 1
    else:
        return 0

def getNDCG(ranklist, K, positive_item):
    if positive_item in ranklist[:K]:
        ranking_of_positive_item = np.where(ranklist == positive_item)[0][0]
        return math.log(2)/math.log(2+ranking_of_positive_item)
    else:
        return 0
```

Listing 5: Implementation of metrics used to evaluate a ranklist of items

4.4 Training implementation

Then, I developped two big functions to evaluate and to train my models respectively, for several models in one go. This is the purpose of the first parameter `models`, which is a list of Keras models. Thanks to these functions I can train several models on the go instead of training them one by one sequentially.

The first function `evaluate_models()` run the evaluation of models included in the list set as argument by performing all the evaluation process I just explained above. It takes as other arguments `test_set` and `K`. It calls the above function `rank()` that returns the ranklist of the 100 test items and then it computes the metrics HR and NDCG from the previous snippet. This function `evaluate_models()` returns the list of HR and NDCG averages for every model passed as argument.

The second function `train_models()` trains the models in argument over a number of epochs, which is specified as a parameter. It calls itself the previous function `evaluate_models()` after each epoch and returns several lists storing every metric after each epoch. In the metrics returned there is also the training loss, which may be interesting for plotting the learning curve.

```python
def rank(item_scores):
    list_item_scores = item_scores.tolist()
    ranklist = sorted(list_item_scores, key=lambda item_score: item_score[1], reverse=True)
    ranklist = np.array(ranklist)[:,0].astype('int64')
    return ranklist

def evaluate_models(models, test_set, K):
    hits, ndcgs = [], []
    users = np.array([user for user in range(len(test_set)) for i in range(100)])
    items = test_set.reshape(-1,)
    for model in models:
        hits_model, ndcgs_model = [], []
        predictions = model.predict(x=[users, items], batch_size=len(test_set), verbose=0)
        map_item_scores = np.concatenate((items.reshape((100*len(test_set), 1)), predictions), axis=1)
        for user in range(len(test_set)):
            ranklist_items = rank(item_scores=map_item_scores[100*user:100*(user+1)])
            positive_item = items[100*user]
            hr = getHitRatio(ranklist=ranklist_items, K=K, positive_item=positive_item)
            ndcg = getNDCG(ranklist=ranklist_items, K=K, positive_item=positive_item)
            hits_model.append(hr)
            ndcgs_model.append(ndcg)
        hits.append(np.array(hits_model).mean())
        ndcgs.append(np.array(ndcgs_model).mean())
    return hits, ndcgs

def train_models(models, train_features, train_labels, test_set, batch_size, epochs, K, verbose=2):
    first_hits, first_ndcgs = evaluate_models(models=models, test_set=test_set, K=K)
    first_losses = []
    for model in models:
        loss = model.evaluate(x=train_features, y=train_labels, batch_size=batch_size, verbose=0)
        first_losses.append(loss)
    losses, hits, ndcgs = [first_losses], [first_hits], [first_ndcgs]
    for e in range(epochs):
        print("\nEpoch n°{}/{}".format(e+1, epochs))
        losses_of_this_epoch = []
        for model in models:
            history = model.fit(x=train_features, y=train_labels, batch_size=batch_size, epochs=1,
            ↪ verbose=verbose, shuffle=True)
            losses_of_this_epoch.append(history.history["loss"][0])
        hits_of_this_epoch, ndcgs_of_this_epoch = evaluate_models(models=models, test_set=test_set,
        ↪ K=K)
        hits.append(hits_of_this_epoch)
        ndcgs.append(ndcgs_of_this_epoch)
        losses.append(losses_of_this_epoch)
    return np.array(hits), np.array(ndcgs), np.array(losses)
```

Listing 6: Functions used to train and evaluate models

4.5 Airline Travel Dataset

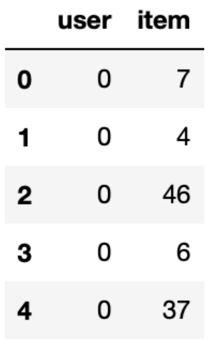
Amadeus IT Group is a leading IT provider for the global travel and tourism industry. It uses a lot of recommender systems and it develops solutions in booking systems. We deal with a subset of their giant database. In this dataset, the users are the customers of an airline and the items denote trip destinations by ID. The purpose of such a recommender system is to recommend to its customers some destinations they are likely to choose for their next trip according to those they already travelled to. The data has been anonymized before and we do not have the table matching destinations and IDs to ensure data privacy. As you can see on the statistics in Table 2 just below, this dataset is very small compared to MovieLens and Pinterest so we may not expect the same performance. It contains only

138 users which is around 100 times less than Pinterest for example, and the number of interactions is very small too. On the other hand the number of items is comparable to MovieLens.

Dataset	**Interactions#**	**Users#**	**Items#**
Airline Travel DS	28,369	138	3,842

Table 2: Statistics of the dataset Airline Travel

The dataset is composed of many pairs (u, i) where u and i are a customer ID and a destination ID respectively. It is made with implicit feedback, so every interaction included in the dataset is a positive interaction and it represents a trip that has really been made by the customer. That is why there is no third column because the score of every pair (u, i) is equal to 1, we need to add this third column in the processing of the dataset.



	user	item
0	0	7
1	0	4
2	0	46
3	0	6
4	0	37

Figure 10: Head of dataset from Airline Travel

This dataset is loaded in one single file and it is not split between train set and the test set has not been prepared yet like it was the case with MovieLens and Pinterest, thus I need to prepare it by myself and to randomly select negative entries.

5 Experiments

The objective of this section is to conduct some experiments on the models and to present the results. These experiments often consist in analysing the behavior of each model when we vary one parameters while keeping all the other fixed in order to highlight its influence on the performance. The parameters that we can study their influence are :

- The number of epochs in training
- The sampling ratio in the dataset
- The number of predictive factors
- The factor K used to compute the evaluation metrics Hit Ratio and NDCG
- The architecture of the MLP model : the number of layers

Without any special mention, we train the models on datasets containing 4 negative interactions per positive interaction, thus the sampling ratio is $N = 4$. Trainings are performed by a Batch Gradient Descent with a batch size of $2^{13} = 8192$ which has empirically proven to be a good trade-off between the Stochastic Gradient Descent and the Batch Gradient Descent. We employ three hidden layers for MLP and NeuMF models with a tower structure. Usually, the number of predictive factors is equal to 8 and we evaluate models with $K = 10$ in the ranklist. We plot the graphics as similar as possible with those in the first paper, with the same colors and the same markers for each model, so that we can compare easily. It is worth noting that the authors of second paper [2] varied the embedding dimension (*i.e.* the number of latent factors) instead of varying predictive factors like they did in the first paper. We decided to vary the number of predictive factors because it may have more influence on the performance. As it is mentioned in the introduction, you can find the code of every experiment in the notebook in my GitHub repository, the link is just below the introduction on page 2. We will end this part by training the models on the Airline Travel Dataset.

5.1 Experiment 1: Training of models and comparison of performance

The first experiment is to train the 4 models on the datasets and to compare their performance. We are going to train them over 10 epochs, which enable the model parameters to converge without spending hours and hours on each experiment. We train them with the same number of predictive factors which is the default value of 8 in order to get comparable results. With such a number of epochs, the training take respectively around 6 min and 16 min on MovieLens and Pinterest which is acceptable. The results on MovieLens and Pinterest are reported respectively in Figure 11 and Figure 12.

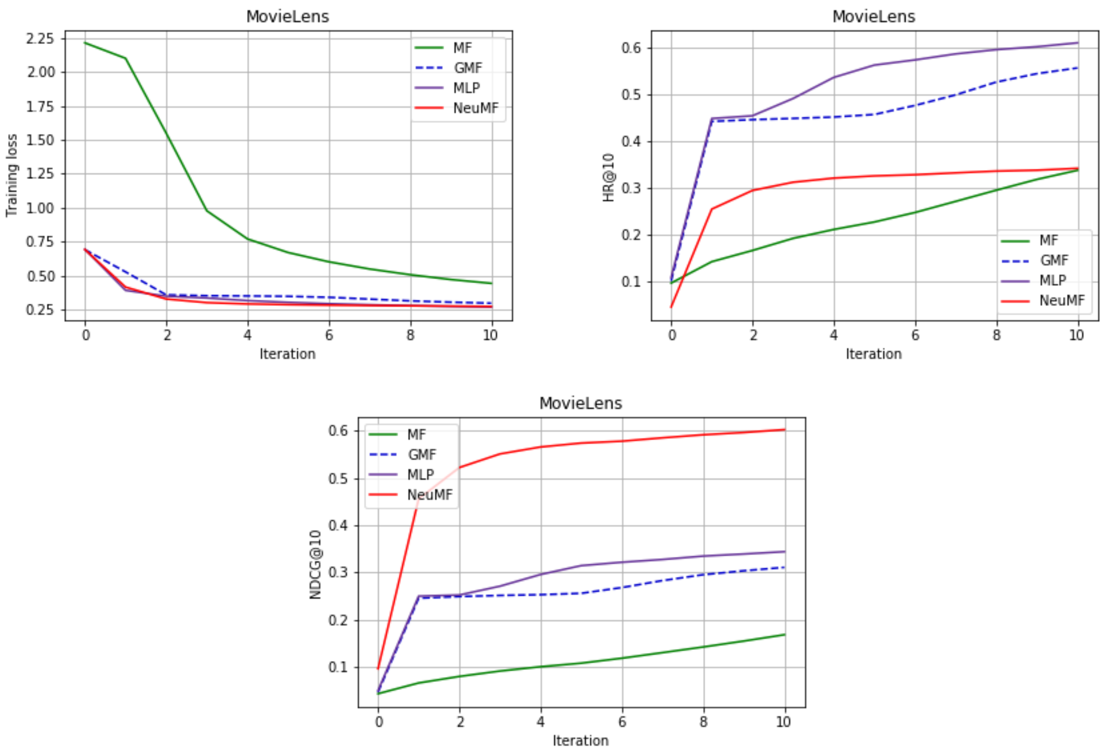


Figure 11: Evaluation of NCF methods over epochs on MovieLens (factors=8)

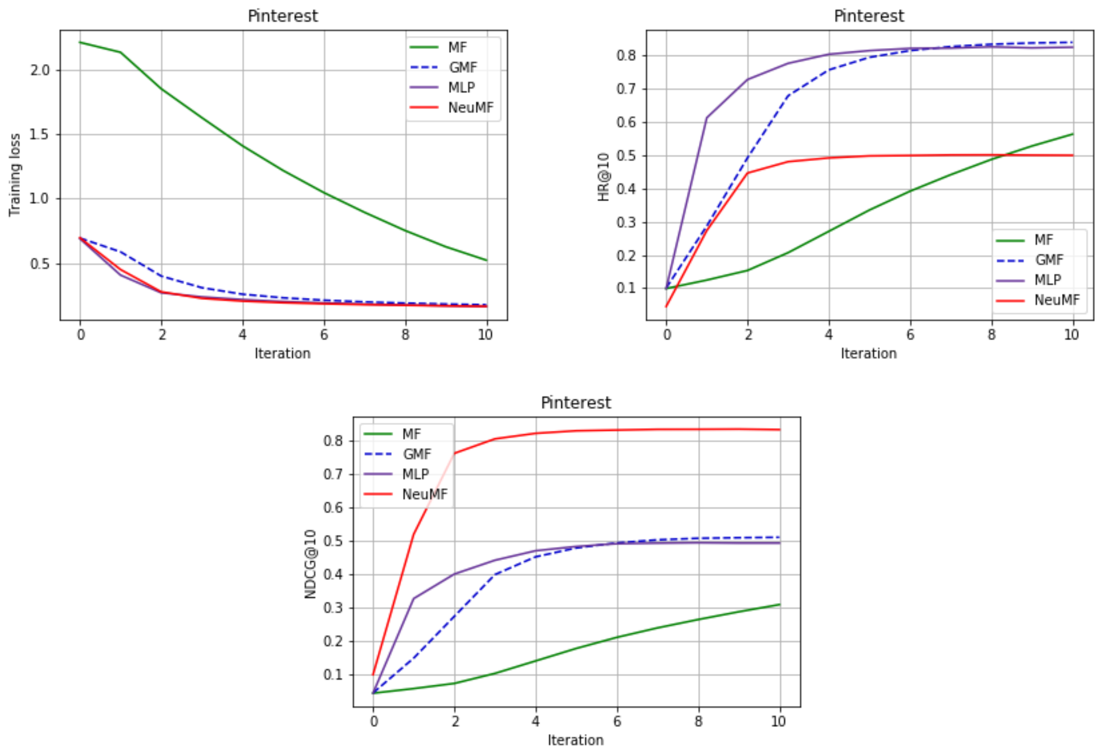


Figure 12: Evaluation of NCF methods over epochs on Pinterest (factors=8)

Contrary to the conclusions of the second paper, we observe here that MF largely under-performs all other models, on all metrics and for both datasets. We recall that this second paper [2] states that MF outperforms every other model if it is well parametrized. That is why we should have a closer look later on the way we train MF and its parameters. As it was expected, the MLP is better than GMF, which is better than MF in all the graphics. This proves the interest of using a learned similarity with more complex models instead of a linear model like MF or GMF. We can see that NeuMF is better than all other models for the metric NDCG, but it is the worst for Hit Ratio, which is a bit strange. The results differ quite a lot from those in the first paper. Regarding the training loss curves, we can conclude that the MF is much longer to converge, because its training loss is higher than the one of any other model. We should implement later the separate pre-training of GMF part and the MLP in the NeuMF to conclude on this model. Anyway, the NDCG better reflects performance of a recommender system because it takes into account the rank of the positive item.

5.2 Experiment 2: Influence of the sampling ratio in the dataset

In this experiment, we vary the parameter N and we observe its influence on performance. N is the number of negative interactions that we sample in the dataset for each positive interaction included in the original dataset. We recall that the dataset originally contains only positive interaction because we work with implicit feedback datasets. So we need to balance the dataset by sampling several negative items for each positive interaction, and the number of these sampled items is equal to N. We experiment N ranging from 1 to 10 included. The default value used by the author of the paper [1] is $N = 4$, so the dataset size is multiplied by 5. N is also called the sampling ratio, because it is the quotient of the number of negative interactions by the number of positive interactions in the whole dataset. This time, we conduct this experiment on the 4 models with 16 predictive factors. Let's see the results reported in Figure 13 just below.

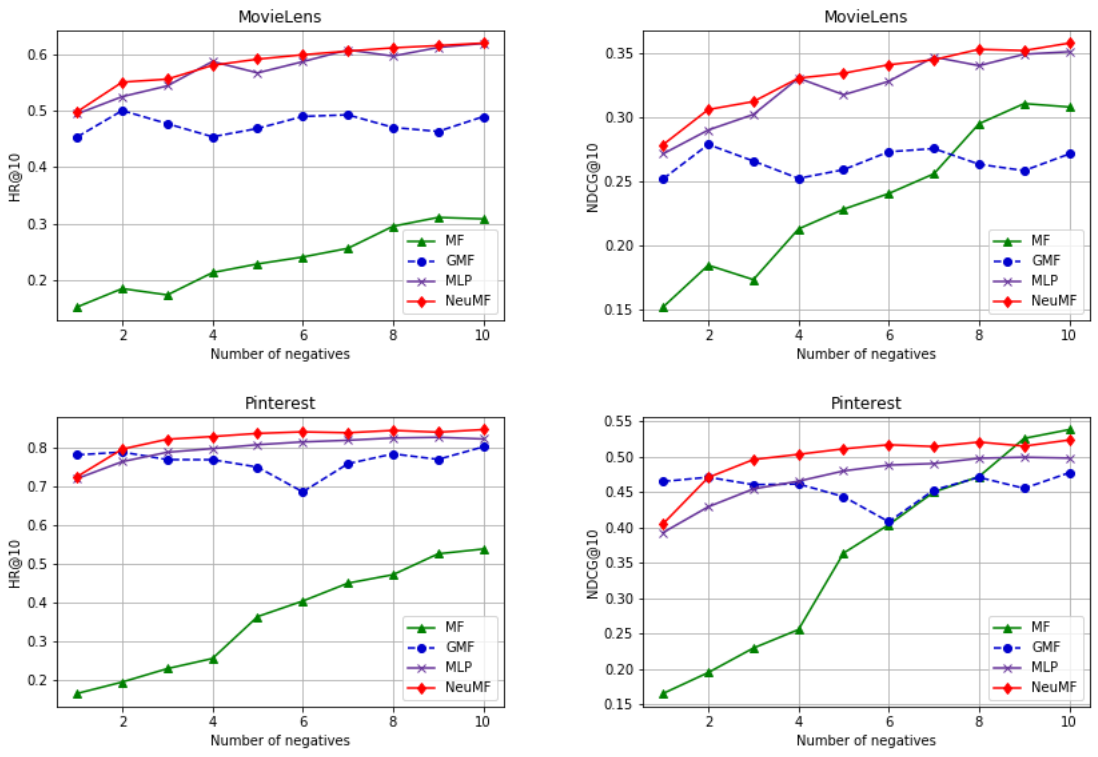


Figure 13: Performance of NCF methods *w.r.t* the number of negative samples per positive interaction on MovieLens and Pinterest (factors=16)

This experiment results in performance that seem to improve slightly when the sampling ratio increases. Still, this trend is not really strong, the performance gain is minor, so we can conclude that using a sampling ratio of 4 as the default value is a good choice, it is not an extreme value. Moreover, we have to notice that increasing N multiplies also the dataset size by N, and thus it is very long to train models for high values of N. On these graphics we might notice that NeuMF outperforms every other model for both metrics and on both dataset, which is consistent with the conclusion from the first paper.

5.3 Experiment 3: The number of predictive factors

Let's draw a conclusion about the influence of predictive factors on the performance of a model. Predictive factors are the last hidden layer of the model, we set the default value as 8 and we kept this value throughout the project. Now it is time to vary this parameter. Intuitively, he number of predictive factors determines the model's capability. But taking too much factors might bring about over-fitting and a lack of generalization of the model.

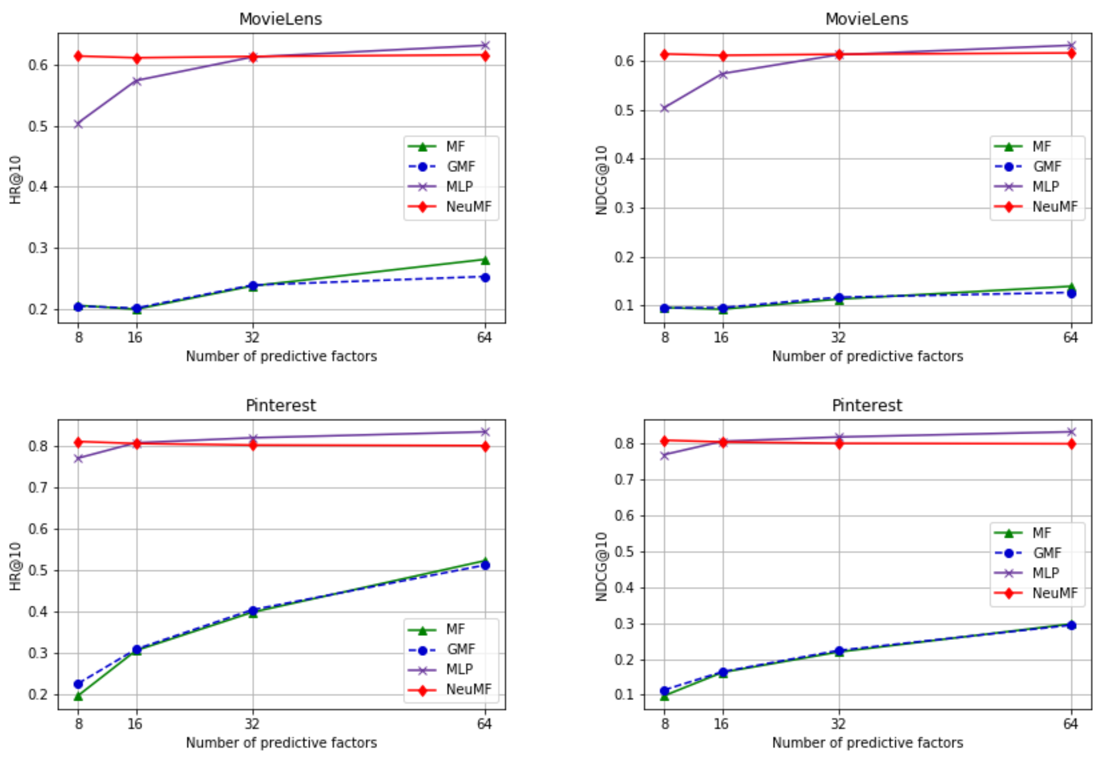


Figure 14: Performance of NCF methods $w.r.t$ the number of predictive factors on MovieLens and Pinterest

What we observed on Figure 14 is a real difference between linear models and deep neural networks with learned similarity. A high number of factors seems to bring a serious gain of performance for MF and GMF, which are the linear models, whereas it does not change anything for MLP and NeuMF. The curves of these two models are practically horizontal.

5.4 Experiment 4: The factor K in the evaluation

This experiment consists in analyzing the performance of Top-K recommended lists where the ranking position K ranges from 1 to 10. We recall that Hit Ratio (HR) returns a binary values which is 1 if the positive item belongs to the top-K ranklist, and 0 otherwise. Normalized Discounted Cumulative Gain (NDCG) better measures the performance by returning a value which depends of the rank of this positive item. Like HR does, NDCG returns 0 if the positive item does not belongs to the top-K ranklist.

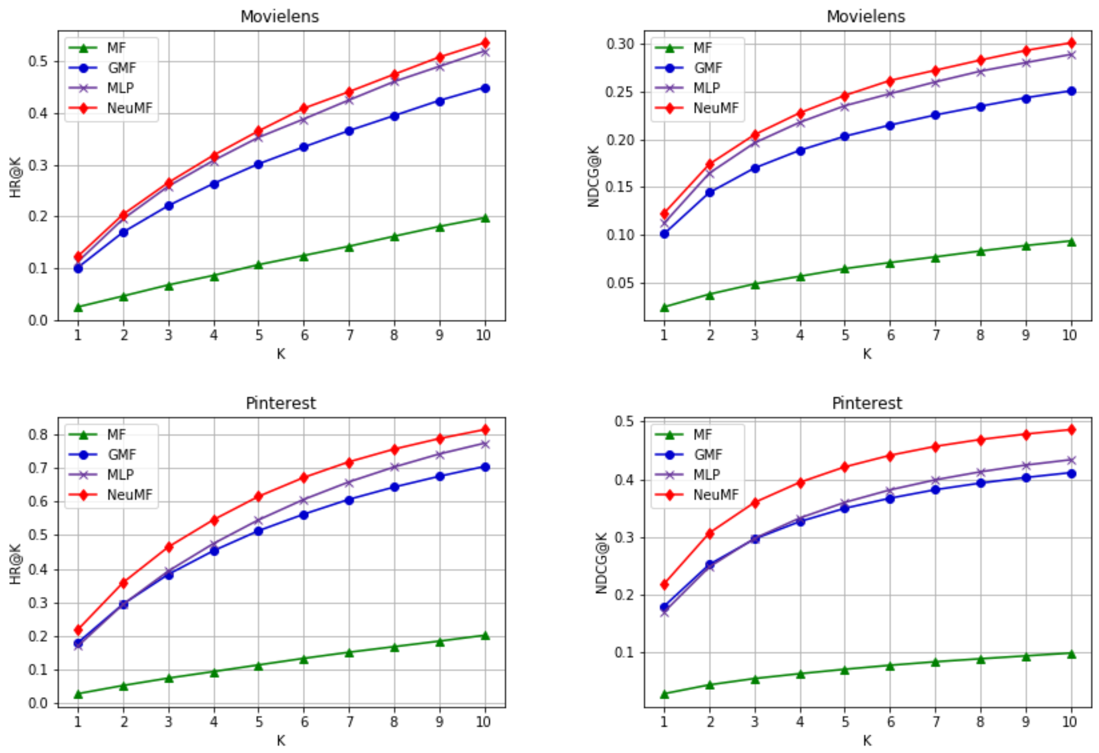


Figure 15: Evaluation of Top-K item recommendation where K ranges from 1 to 10 on MovieLens and Pinterest (factors=8)

The results are reported in Figure 15. As we could expect, the metrics increase with K, for both metrics HR and NDCG, for both datasets MovieLens and Pinterest and for all models. Intuitively, taking a small K is not very interesting because positive items must be ranked among the really top items in the ranklist. For example, if $K = 1$ then both metrics are null each time the positive item is not ranked first in the ranklist, which is almost always the case. Any model could offer good performance even if positive items are not ranked first, but in the Top-5 or Top-10 for example, because recommendations may be pertinent. The greater the factor K is, the more likely the positive item to belong to the Top-K is. The authors of the paper [1] take $K = 10$ for all their experiments and it seems to be a good value. Because taking K higher would not better reflect performance, because a recommendation is not pertinent if the positive item is ranked 30th in the ranklist for example.

5.5 Experiment 5: Architecture of the neural network in MLP

In this experiment we want to reveal the influence of the architecture of the network: the number of layers and the number of units per layer. As we did troughtout the project, we still use a tower pattern, where the bottom layer is the widest and each successive layer has a smaller number of neurons. The number of neurons per layer is completely determined by the number of predictive factors, because we take the powers of 2. It means that every hidden layer has two times more units than its next hidden layer, and we can build the network layer by layer by starting from the predictive factors and by doubling the number of neurons for each layer.

We are going to generate a lot of models having different numbers of predictive factors and different structures of units. We take the following values for predictive factors : 8, 16, 32 and 64. An MLP with n hidden layers is denoted as MLP-n. For example, MLP-4 means that the MLP has 4 hidden layers, and so on. The results are summarized in Table 3 and Table 4. The experiment consists in training $5 \times 4 = 20$ models for each dataset, because there are 5 numbers of hidden layers (from 0 to 4 included) and 4 number of predictive factors. Every model is quickly trained over 3 epochs.

Factors	**MLP-0**	**MLP-1**	**MLP-2**	**MLP-3**	**MLP-4**
MovieLens					
8	0.451	0.449	0.465	0.505	0.578
16	0.449	0.443	0.512	0.565	0.606
32	0.448	0.452	0.539	0.612	0.633
64	0.451	0.482	0.601	0.629	0.639
Pinterest					
8	0.274	0.364	0.734	0.773	0.796
16	0.273	0.445	0.772	0.800	0.824
32	0.273	0.669	0.797	0.821	0.834
64	0.272	0.757	0.819	0.834	0.840

Table 3: HR@10 of MLP with different structures on MovieLens and Pinterest

Factors	**MLP-0**	**MLP-1**	**MLP-2**	**MLP-3**	**MLP-4**
MovieLens					
8	0.251	0.251	0.258	0.277	0.328
16	0.250	0.250	0.285	0.315	0.345
32	0.250	0.251	0.301	0.351	0.364
64	0.249	0.267	0.338	0.361	0.367
Pinterest					
8	0.141	0.185	0.405	0.440	0.461
16	0.140	0.232	0.436	0.469	0.493
32	0.140	0.366	0.467	0.491	0.499
64	0.139	0.423	0.490	0.501	0.507

Table 4: NDCG@10 of MLP with different structures on MovieLens and Pinterest

The results are quite similar with those in the paper. As you can observe, even for models with the same capability (*i.e.* the same number of predictive factors), stacking more layers is beneficial to performance. This result is encouraging, indicating the effectiveness of using deep models for collaborative recommendation. We may attribute the improvement to the non-linearities brought by stacking more non-linear layers. For MLP-0 that has no hidden layers (*i.e.*, the embedding layer is directly projected to predictions), the performance is very weak. This confirms that simply concatenating user and item latent vectors is insufficient for modelling their feature interactions. We observe also that performance improve when the number of predictive factors increase a bit. You can observe that by focusing on any column. This confirms what the experiment 3 has revealed.

5.6 Experiment 6 : Training on the Airline Travel Dataset

Finally, we experiment in this part the four models on the Airline Travel Dataset that we presented in the previous section. As we mentioned, the dataset is made of only one single csv file containing the list of positive interactions. Thus, I developped a specific function for this dataset which imports it. I do not include this function in this report but you can find it in the notebook in my GitHub repository in the introduction. Since the dataset is tiny, we used a smaller sampling ratio of 2 instead of 4. It should prevent too much overlap between positive and negative in the train set.

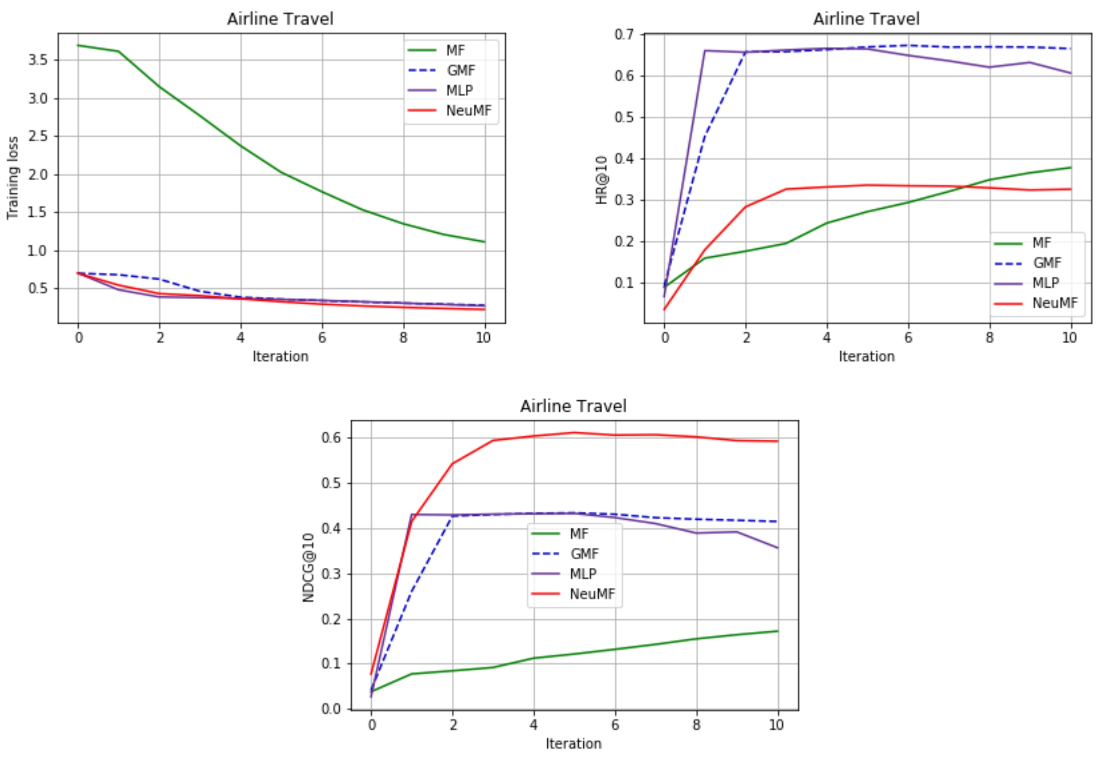


Figure 16: Evaluation of NCF methods over epochs on Airline Travel dataset (factors=8)

Conclusion and Future Work

As a conclusion on this project, we can state that we could successfully reproduce all the experiments conducted in the first paper [1], with quite similar and consistent results. However, the results we get do not go along with what is stated in the second paper [2]. The results we obtain seem to show that the best performance is achieved by the most complex models, thus the MLP and the NeuMF.

The authors of the second paper have experimented the learning of a dot product. They have sampled random embeddings for building their dataset, they have computed the dot product between all pairs and they have had a Gaussian noise to get the train labels. They have tried to approximate the dot product with an MLP but they demonstrated that the required size of the dataset scales polynomially with the wanted error. This experiment shows that even if the MLP is a universal function approximator, it is really difficult to fit some functions. They also concluded through their other experiment that MF outperforms the learned models but our results are contradictory with their conclusion. We should go further into experiments as a future work.

The future developments of this project may include the pre-training of the MLP part and the GMF part as initialization of the NeuMF model, which has turned out to be even more efficient than every method present in this work. This would just require a bit more work on the code. We will also experiment the trade-off α, this hyper-parameter determines the trade-off between the MLP part and the GMF part. We arbitrarily set it to 0.5 as the default value but it would be interesting to observe the behaviour of NeuMF when we slightly change it. An other improvement of this work would be to consider MovieLens as explicit feedback instead of converting it to implicit feedback, because it is originally an explicit feedback dataset where scores range from 1 to 5. We can make the models estimating the original score between 1 and 5 and evaluate them with other appropriate metrics. And finally, we will try these models on other datasets, maybe on the whole datasets MovieLens and Pinterest because actually the datasets we worked with are just subsets of these datasets, or on the Netflix Prize dataset. We should conduct more experiments on the industrial dataset from Amadeus, maybe with the whole dataset too because this one was tiny compared to the others.

Acknowledgement

Finally, I would like to thank Amine DADOUN (Phd student at Eurecom and Amadeus) and Raphaël TRONCY (Professor at Eurecom - Data Science Department) for having supervised and assisted me throughout the project. Through them I could lead the project quite smoothly and I improved my skills in Data Science.

References

[1] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering, 2017.

[2] Steffen Rendle, Walid Krichene, Li Zhang, and John Anderson. Neural collaborative filtering vs. matrix factorization revisited, 2020.