Implementation of Neural Collaborative Filtering

Jixuan Sun, Xiang Qu, Zhifei Dou, Siyan Liu

Abstract

There are widespread applications of recommendation systems in the world. Neural networks have been deployed to enhance the performance of recommender models that use the collaborative filtering technique. The project aims to assess the effectiveness and generalizability of generalized matrix factorization (GMF), multi-layer perceptron (MLP), and neural matrix factorization (NeuMF) models through measuring their mean square error loss when applied to two datasets. The project will empirically tune the hyperparameters for each model using MovieLens data and evaluate their generalizability by applying the tuned models on Amazon's user rating data. Meanwhile, SGD and Adam optimizer will have their performance compared with each other under the same set of hyperparameters. The result shows that NeuMF has the least loss when tested on the Amazon dataset, followed by MLP, while GMF has the highest loss due to the model's linear nature. Adam optimizer produces less loss than SGD. The performance of the three models matches with that observed in the main reference paper, concluding that recommendation performance is better for models that involve neural networks.

Project Implementation GitHub: https://github.com/ZhifeiDou/MIE424_Final_Project.git

1 Introduction

RECOMMENDATION systems can make predictions to users' preferences for items based on known user preferences and item characteristics. The systems make a significant difference in user engagement because they can quickly find the best options out of many [1]. Their applications are widespread, from shopping platforms like Amazon enhancing the users' shopping experiences through tailored product suggestions, to streaming services such as Netflix revolutionizing content consumption with personalized media recommendations [2]. By analyzing data in detail, recommendation systems can relatively accurately predict the user's preference, making their suggestions more tailored to each person's interest.

Motivated by our interest in recommendation systems, the team explored a broad array of related papers and their applications. In this project, we aim to reconstruct and modify the Neural Collaborative Filtering (NCF) models, as proposed by He *et al.* [1], to validate their performance in predicting a user's rating for an item. Neural Matrix Factorization (NeuMF), a model that belongs to the broader category of NCF models, stands out for its ability to simultaneously generalize the matrix factorization (MF) model, while incorporating nonlinearities through a multi-layer perceptron (MLP). The team will first develop the generalized matrix factorization (GMF) and MLP models and perform hyperparameter tuning. Subsequently, we will integrate these models to create the NeuMF model, as described in the referenced paper. We then plan to conduct experiments to determine whether the combined model indeed performs better, which is measured by achieving a smaller loss. Additionally, we will evaluate the impact of implementing different optimizers such as stochastic gradient descent (SGD) and Adam for the recommendation systems.

2 Related Work

The referenced paper mainly explores the integration of the GMF model with the MLP model to formulate the NeuMF model. However, it lacks comprehensive details on the constituent GMF and MLP models. Therefore, the team needs to consult supplementary papers to acquire a thorough understanding of both the GMF and MLP models before proceeding.

GMF allows the representation of user and item embeddings in a latent space. Traditional MF applies the inner product between two matrices to represent the similarities between two entities (e.g., users and movies) [3]. This is an instance of the user-item interaction matrix. However, the inner product is limiting the expressiveness of the model. Thus, the authors of our main reference paper, He *et al.*, developed a method to learn the user-item interaction function from the data to increase the expressiveness of the model, by having a neural layer that generalizes the MF model to become GMF.

MLP, on the other hand, is a type of artificial neural network composed of multiple layers of neurons, utilizing non-linear activation functions to enable the modeling of complex relationships between input and output variables [4]. This characteristic allows the model to capture complex patterns and interactions between features that are not possible with linear models. As a result, MLP is highly regarded for its ability to enhance the feature representation and predictive power of neural network-based models. However, the generalization ability of this model is unknown. Consequently, there is an opportunity to combine the advantages of both MF and MLP to create Neural Matrix Factorization (NeuMF).

According to the referenced paper by He *et al*, GMF applies a linear kernel to model the feature interactions while MLP applies a non-linear kernel to learn the interaction function [1]. To achieve a better model performance, the next step in the paper is to fuse the GMF and MLP models. Thus, the team will work on reconstructing the GMF and MLP models and combining these two models to form the NeuMF model. The standard structure of a NeuMF model is demonstrated in Figure 1 below.

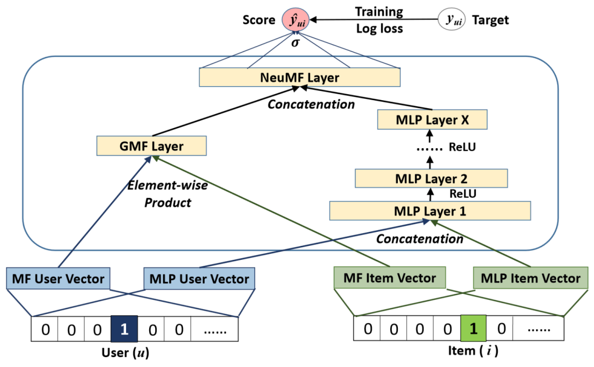


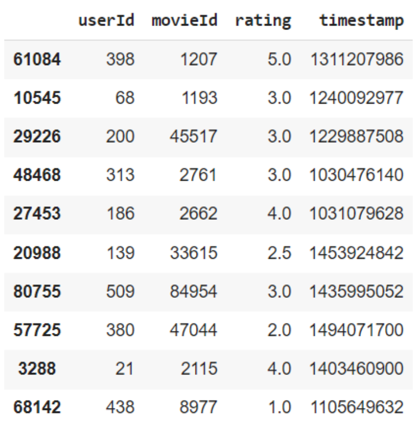
Figure 1. Neural Matrix Factorization Model [1]

Thus, the primary focus of this project is to evaluate these three types of NCF models in a new context by modifying the models from binary classification models to regression models. In the original paper, NeuMF was used to predict whether a user and an item had an interaction, a type of binary classification. In our project, we used it to predict the rating using a scale from 1 to 5 between a user and an item. Using the model on a regression problem can give us more insight into its effectiveness over the more primitive models.

3 Datasets

The major training data utilized by the team is sourced from MovieLens, which is a renowned web-based recommender system and virtual community dedicated to suggesting movies based on user preferences [5]. It employs collaborative filtering techniques, analyzing members' movie ratings and reviews to make tailored recommendations. This dataset comprises approximately 11 million ratings spanning around 8,500 movies.

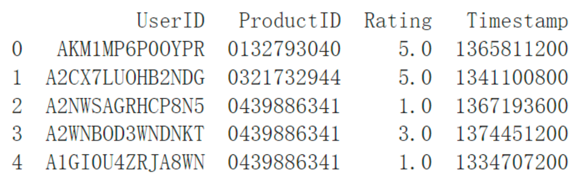
Specifically, the team is utilizing the MovieLens Latest Datasets [6]. This dataset encapsulates 5-star ratings and free-text tagging activities extracted from MovieLens, providing insights into user interactions with movies. It encompasses a total of 100,004 ratings and 1,296 tag applications across 9,125 movies. The training data is stored within the *rating.csv* file, featuring columns for userId, movieId, rating, and timestamp, as illustrated in Figure 2.



	userId	movieId	rating	timestamp
61084	398	1207	5.0	1311207986
10545	68	1193	3.0	1240092977
29226	200	45517	3.0	1229887508
48468	313	2761	3.0	1030476140
27453	186	2662	4.0	1031079628
20988	139	33615	2.5	1453924842
80755	509	84954	3.0	1435995052
57725	380	47044	2.0	1494071700
3288	21	2115	4.0	1403460900
68142	438	8977	1.0	1105649632

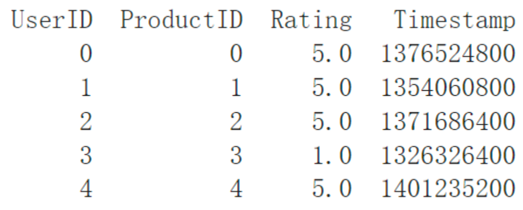
Figure 2. Sample Data of MovieLens Latest Datasets

The second dataset utilized in this project is the Amazon Reviews dataset. This implicit rating data contains Amazon's user rating for Amazon's electronics products [7]. The dataset is employed to assess the generalization of the modified regression recommendation system. Similar to the MovieLens dataset, the Amazon Reviews dataset consists of columns for UserID and ProductID as shown in Figure 3. The user data and the product data within the dataset are stored as a unique ID consisting of letters and numbers. Hence, the team has mapped each of the user ID and product ID into a unique integer for the training purpose as demonstrated in Figure 4. Moreover, the rating column identifies the user's rating of the products. A rating between 1 to 5 indicates how satisfied the user is with the product. The team will analyze the model performances using this dataset and compare the results with those results reported with the MovieLens dataset.



```
          UserID      ProductID  Rating   Timestamp
0   AKM1MP6P0OYPR   0132793040     5.0  1365811200
1  A2CX7LUOHB2NDG   0321732944     5.0  1341100800
2  A2NWSAGRHCP8N5   0439886341     1.0  1367193600
3  A2WNBOD3WNDNKT   0439886341     3.0  1374451200
4   A1GI0U4ZRJA8WN  0439886341     1.0  1334707200
```

Figure 3. Sample Data of Amazon Review Datasets



```
UserID  ProductID  Rating   Timestamp
     0          0     5.0  1376524800
     1          1     5.0  1354060800
     2          2     5.0  1371686400
     3          3     1.0  1326326400
     4          4     5.0  1401235200
```

Figure 4. Sample Data of Amazon Review Datasets After Processing

4 Methods

4.1 Loss Function

As we discussed, the team has modified the models to a regression model for predicting the user's rating for an item. Hence, the log loss (a.k.a binary entropy loss) from He's paper will not be suitable anymore due to its binary property. The team has changed the loss function to MSE loss which can be defined as:

$$MSE\ Loss = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

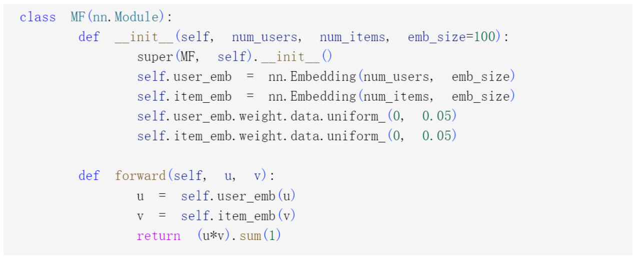
Thus, the objective function will be a minimization function demonstrated as the following:

$$min\ \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

The MSE loss is a reasonable choice since it is differentiable, which allows for the application of gradient descent and gradient descent's extensions, such as SGD and Adam.

4.2 Model

By following what the paper has outlined, the team has successfully reconstructed the GMF, MLP, and NeuMF models with Pytorch. For all three models, the team has modified them by removing the last sigmoid activation function to fully connect the last layer with the output node; thus, accommodating our regression problem.



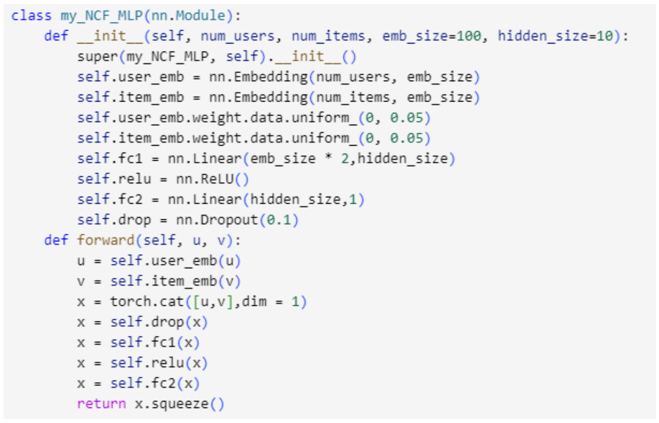
```
class  MF(nn.Module):
        def  __init__(self,  num_users,  num_items,  emb_size=100):
                super(MF,  self).__init__()
                self.user_emb  =  nn.Embedding(num_users,  emb_size)
                self.item_emb  =  nn.Embedding(num_items,  emb_size)
                self.user_emb.weight.data.uniform_(0,  0.05)
                self.item_emb.weight.data.uniform_(0,  0.05)

        def  forward(self,  u,  v):
                u  =  self.user_emb(u)
                v  =  self.item_emb(v)
                return  (u*v).sum(1)
```

Figure 5. GMF Model in Pytorch

As demonstrated in Figure 5, the team has reconstructed a GMF model with two latent factors denoted as "user_emb" and "item_emb", with 100 default embedding size. Once the model receives the input userID and itemID, the corresponding column within latent factors will be extracted as parameters. Both latent factors will be initialized with values drawn from a uniform distribution between 0 and 0.03. The output for the model is the element-wise dot product between two latent factors, which will reflect the user's rating of a specific item. Accordingly, the optimization objective function for GMF is:

$$\hat{y}_{ui} = \sum_{k=1}^{K} p_{uk}q_{ik} = p_u \odot q_i$$

where **p** and **q** denote the latent vector for user u and item i , K represent the dimension of the latent space. This objective function is continuous and convex since $p_{uk}, q_{ik} \in R\ \forall u, i, k$ and the second derivative for this function is always 0.



```
class my_NCF_MLP(nn.Module):
    def __init__(self, num_users, num_items, emb_size=100, hidden_size=10):
        super(my_NCF_MLP, self).__init__()
        self.user_emb = nn.Embedding(num_users, emb_size)
        self.item_emb = nn.Embedding(num_items, emb_size)
        self.user_emb.weight.data.uniform_(0, 0.05)
        self.item_emb.weight.data.uniform_(0, 0.05)
        self.fc1 = nn.Linear(emb_size * 2,hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size,1)
        self.drop = nn.Dropout(0.1)
    def forward(self, u, v):
        u = self.user_emb(u)
        v = self.item_emb(v)
        x = torch.cat([u,v],dim = 1)
        x = self.drop(x)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x.squeeze()
```

Figure 6. MLP Model in Pytorch

Next, the team has reconstructed the MLP model, which involves neural layers, as demonstrated in Figure 6. For the MLP model, two latent factors are also involved as parameters representing each user and item. Then, the model takes the concatenation of two latent factors and inputs the concatenated latent factors into a neural network with two fully connected layers with a ReLU activation function.

The ReLU activation function breaks the linearity of the model, thus increasing the expressiveness. Meanwhile, a drop-out layer is utilized to prevent overfitting. Similar to GMF, the output is a vector in which each value in the vector reflects a user's rating of an item. Hence, the optimization objective functions are:

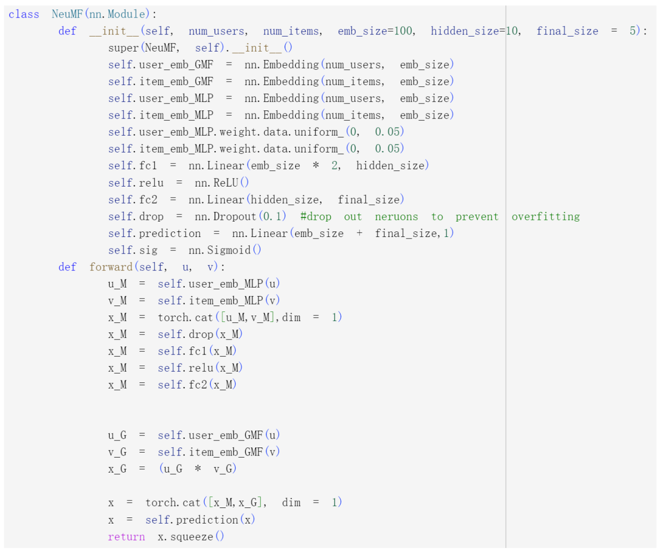
$$z_1 = cat(p_u, q_i)$$
$$z_2 = \phi_2(z_1) = a(W_1^T z_1 + b_1)$$
$$\hat{y}_{ui} = \phi_3(z_2) = W_2^T z_2 + b_2$$

which is equivalent to:

$$\hat{y}_{ui} = W_2^T a(W_1^T cat(p_u, q_i) + b_1) + b_2$$

where $cat()$ represents the concatenation of user and item latent factors. W_i^T represent the weight parameter matrices for neural layers, b_i denote bias terms, and a is the ReLU activation function between layers. This function is a continuous but non-convex function since the ReLU activation function's second derivative changes from 0 to undefined.

After that, the team reconstructed the NeuMF model by combining the GMF and MLP models. The model is demonstrated in the following Figure 7:



```
class NeuMF(nn.Module):
    def __init__(self, num_users, num_items, emb_size=100, hidden_size=10, final_size = 5):
        super(NeuMF, self).__init__()
        self.user_emb_GMF = nn.Embedding(num_users, emb_size)
        self.item_emb_GMF = nn.Embedding(num_items, emb_size)
        self.user_emb_MLP = nn.Embedding(num_users, emb_size)
        self.item_emb_MLP = nn.Embedding(num_items, emb_size)
        self.user_emb_MLP.weight.data.uniform_(0, 0.05)
        self.item_emb_MLP.weight.data.uniform_(0, 0.05)
        self.fc1 = nn.Linear(emb_size * 2, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, final_size)
        self.drop = nn.Dropout(0.1) #drop out neruons to prevent overfitting
        self.prediction = nn.Linear(emb_size + final_size,1)
        self.sig = nn.Sigmoid()
    def forward(self, u, v):
        u_M = self.user_emb_MLP(u)
        v_M = self.item_emb_MLP(v)
        x_M = torch.cat([u_M,v_M],dim = 1)
        x_M = self.drop(x_M)
        x_M = self.fc1(x_M)
        x_M = self.relu(x_M)
        x_M = self.fc2(x_M)


        u_G = self.user_emb_GMF(u)
        v_G = self.item_emb_GMF(v)
        x_G = (u_G * v_G)

        x = torch.cat([x_M,x_G], dim = 1)
        x = self.prediction(x)
        return x.squeeze()
```

Figure 7. NeuMF Model in Pytorch

In the reconstructed NeuMF model, the default embedding size and hidden size remain the same. Another hyperparameter — final size — is added to identify the neural layer's size for MLP within the NeuMF. After the combination, the prediction neural layer for the NeuMF model takes the concatenation of outputs from GMF and MLP. It outputs a single prediction value for each data point which reflects the user's rating of an item. Therefore, the optimization objective function for NeuMF is:

$$\phi_{GMF} = p_u \odot q_i$$
$$\phi_{MLP} = W_2^T a(W_1^T cat(p_u, q_i) + b_1) + b_2$$
$$\hat{y}_{ui} = W_3^T cat(\phi_{GMF}, \phi_{MLP})$$

Since the NeuMF 's function is the concatenation of the GMF and MLP model, its continuous property will remain the same; meanwhile, the convexity from the GMF model will be broken by the activation from the MLP model, resulting in a non-convex function for NeuMF model.

4.3 Training Function

To train the model, the team has utilized a training function named "train_epoch", which takes the MSE loss and utilizes the Adam optimizer for optimization. By implementing the Adam optimizer, the team can benefit from utilizing moment and adjusted learning rate to obtain better performance for models with non-linear objective functions [8]. According to the experiment performed by the introducer of Adam optimizer, Adam indeed outperformed the traditional SGD when training the non-linear neural network model such as convolutional neural network (CNN) models [9]. For testing purposes, the team has also implemented the SGD optimizer as a comparison with the Adam optimizer. The result of this comparison will be introduced in the following section.

5 Experiment and Result

5.1 Experiment Setup

Experiments mentioned in the next session are conducted with Google Colab. All the experiments were run on the CPU. The packages used in experiments are pandas, numpy, pytorch and matplotlib. Due to the limitation of RAM usage, the team randomly selected records from the MovieLens dataset to form training and testing sets accordingly, which are 100,837 data points in total. The ratio of training sets and testing sets is 75:25.

5.2 Hyperparameter Tuning

The team tuned hyperparameters using the MovieLens dataset. GMF, MLP, and NeuMF have different structures, so they have different hyperparameters that can affect the model's performance. The hyperparameters of each model are tuned separately. The list of hyperparameters of each model is below.

Table 1. List of Tuned Hyperparameters for All Models

Model	Hyperparameters
GMF	● learning rate ● weight decay ● embedding size ● number of training epochs
MLP	● learning rate ● hidden layer size ● embedding size ● weight decay ● number of training epochs
NeuMF	● learning rate ● hidden layer size ● embedding size ● final size ● weight decay ● number of training epochs

The team first prioritizes these hyperparameters based on the structure of each model. Then the team implemented the sequential hyperparameter method and empirical testing, which is tuning hyperparameters one by one with a list of values and using the best-tuned values for each hyperparameter as the basis for tuning the next. The order of hyperparameter tuning is slightly different for each model due to the differences in the model structure.

All the models started with tuning the learning rate and ended with tuning the number of training epochs. Although all models were trained using the Adam optimizer, they still needed a reasonable initial learning rate to achieve better performance with the implementation of an adaptive learning rate. All models shared the same list of initial values for learning rate, which include commonly used values for training: [0.000001, 0.00001, 0.0001, 0.001, 0.01, 0.1, 0.9] and models also share the same list of initial values for weight decay, which is [0.000001, 0.00001, 0.0001, 0.001, 0.01]. The initial values of other hyperparameters are selected based on the model's complexity and input size.

The team iteratively trained each model with a value in the initial value list and narrowed down the value range with lower testing loss until the team found the hyperparameter value with the lowest testing loss.

5.2.1 Order of Hyperparameter Tuning for GMF

After tuning the learning rate, the team tuned weight decay, embedding size and number of training epochs sequentially. Weight decay is a form of regularization that helps prevent overfitting by penalizing large weights in the model. And the embedding size determines the dimensionality of the latent factors. Tuning weight decay before the embedding size can help avoid overfitting with a smaller embedding size which can make training faster and more efficient. The number of training epochs was tuned lastly to find the best time for the model to stop training.

5.2.2 Order of Hyperparameter Tuning for MLP

MLP has hidden layers which makes it different from GMF. Hidden layer size and embedding size can impact the model capacity together. Thus for MLP, the team tuned hidden layer size first and then the embedding size before tuning the weight decay. The purpose was to find the right model size before implementing the regularization and improve tuning efficiency.

5.2.3 Order of Hyperparameter Tuning for NeuMF

NeuMF has the "final size" parameter, which is the output of the GMF inside NeuMF. Similar to tuning MLP, the team tuned the final size before tuning the weight decay and after tuning the hidden layer size and embedding size.

5.2.4 Hyperparameter Tuning Results

Following the procedures indicated above, the team successfully tuned all the hyperparameters and obtained the testing loss for each model. The testing loss for GMF is 0.81, while that of MLP and NeuMF is lower at 0.77 and 0.78, respectively. The results differ from the reference paper slightly by having NeuMF performing slightly worse than MLP. The tuned hyperparameter values for each model are shown in the following tables.

Table 2. Tuned Hyperparameter Values for GMF

Hyperparameter	**Value**
learning rate	0.09
weight decay	0.00004
embedding size	108
number of training epochs	54

Table 3. Tuned Hyperparameter Values for MLP

Hyperparameter	**Value**
learning rate	0.09
hidden layer size	30
embedding size	47
weight decay	0.000006
number of training epochs	45

Table 4. Tuned Hyperparameter Values for NeuMF

Hyperparameter	**Value**
learning rate	0.03
hidden layer size	85
embedding size	250
final size	7
weight decay	0.00005
number of training epochs	40

5.3 Compare with Amazon Review Dataset

After the team finished hyperparameter tuning, we tested all models with their best hyperparameter setting and the Amazon Review Dataset mentioned in section 3 for testing the generalization ability for NCF models in a regression setting. Due to the RAM limitation, the team could not train and test the model with the entire dataset. Subsequently, the team withdrew 5% of the original dataset, 391,224 data points, which is almost 4 times the MovieLens subset.

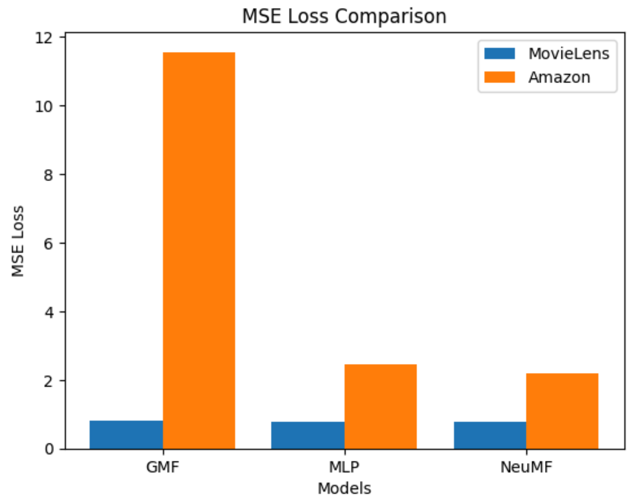


Figure 8. MSE Loss Across Two Datasets

Shown in Figure 8, with the best hyperparameters setting, the team has obtained 11.557 MSE loss for the GMF model, 2.455 for MLP, and 2.198 for NeuMF. According to the MSE loss, we can suggest that the GMF model with regression setting has poor generalization ability, which might be caused by the linearity property of GMF. In terms of MLP and NeuMF, the loss went slightly worse but was still acceptable; thus, identifying a better generalization ability might be caused by the non-linearity property and more parameters.

5.4 Experiment for SGD and Adam

The team has also implemented the experiment with an SGD optimizer to validate the superiority of the Adam optimizer. Accordingly, the team has performed hyperparameter tuning and tested the NeuMF model with SGD, resulting in a worse testing loss compared to the same metric trained with Adam. The MSE loss of SGD was 1.09 and that of Adam was only 0.78, with a difference of 0.31. Hence, the team would suggest that the adaptive learning rate technique adopted by Adam helps to achieve a better optimized minimum value. Please see the original code uploaded to GitHub for detailed implementation.

6 Conclusion and Future Work

Throughout this project, the team evaluated the performance of three NCF models — GMF, MLP, and NeuMF against two datasets. While we evaluated the same models as in the referred paper, we have used the models to take on regression tasks instead of binary classification.

After comparing the MSE loss across the three models on the Amazon dataset, we find that the NeuMF model did indeed produce the lowest level of loss, which agrees with the original paper's results that NeuMF has the best performance. Meanwhile, in terms of generalization, MLP and NeuMF have demonstrated good generalization ability with relatively low MSE loss. On the other hand, GMF, as a linear model, cannot generalize well when doing regression tasks, which is reflected in its high MSE loss in the experiment.

From the experiment between SGD and Adam optimizers, Adam produced better results. This indicates that having an adaptive learning rate can better optimize the final value of gradient descent, as opposed to pure SGD. Even though the improvement in MSE loss of 0.31 is not as large of a number, it is still significant because having an incremental improvement in accuracy when applied to a large dataset can massively increase the total number of "correct" predictions made.

In the future, the team would solve this rating problem by implementing a multi-class classification model with a softmax activation function in the last layer. Since the regression model's continuous output doesn't exactly reflect the user's rating of an item, a multi-class classification model can directly prove the most possible rating for us. Meanwhile, other than the user-item interaction that can affect the rating, bias for a certain user or item can also impact the rating; hence, additional bias parameters can also be involved in the model for better model performance in the future. Also, the team would examine the model's performance with various GPUs to investigate these models' potential further.

Reference

[1] He, X. et al. (2017) 'Neural collaborative filtering', Proceedings of the 26th International Conference on World Wide Web [Preprint]. doi:10.1145/3038912.3052569.
[2] S. Arora, "Recommendation engines: How Amazon and Netflix are winning the Personalization Battle," Spiceworks, https://www.spiceworks.com/marketing/customer-experience/articles/recommendation-engines-how-amazon-and-netflix-are-winning-the-personalization-battle/ (accessed Apr. 7, 2024).
[3] Bokde, Dheeraj & Girase, Sheetal & Mukhopadhyay, Debajyoti. (2015). Matrix Factorization Model in Collaborative Filtering Algorithms: A Survey. Procedia Computer Science. 49. 10.1016/j.procs.2015.04.237.
[4] "Multilayer perceptrons for classification and regression," Neurocomputing, https://www.sciencedirect.com/science/article/abs/pii/0925231291900235 (accessed Apr. 7, 2024).
[5] "Non-commercial, personalized movie recommendations.," MovieLens, https://movielens.org/ (accessed Mar. 22, 2024).
[6] "Movielens," GroupLens, https://grouplens.org/datasets/movielens/ (accessed Mar. 22, 2024).
[7] S. Anand, "Amazon product reviews," Kaggle, https://www.kaggle.com/datasets/saurav9786/amazon-product-reviews (accessed Apr. 7, 2024).
[8] "Adam¶," Adam - PyTorch 2.2 documentation, https://pytorch.org/docs/stable/generated/torch.optim.Adam.html (accessed Apr. 12, 2024).
[9] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," arXiv.org, https://arxiv.org/abs/1412.6980 (accessed Apr. 12, 2024).

Contributions

Jixuan Sun: drafted introduction, related work and dataset sections, researched reference paper and other relevant literature, prepared datasets, compiled citations, proofread and formatted the whole paper

Xiang Qu: reviewed reference paper, drafted abstract and conclusion sections, drafted parts of introduction and experiment sections, created visuals for experiment results, proofread and formatted the whole paper

Zhifei Dou: researched reference paper and Adam paper, drafted, revised, and finished the methods section, drafted parts of experiment section, prepared and processed datasets, developed code for models, performed experiment between optimizers, proofread report

Siyan Liu: read the primary paper, summarized the primary paper, drafted and revised parts of the experiment section, designed experiments, developed strategies and code for tuning hyperparameters, created visuals for experiment results, proofread and formatted the whole paper