



Bilkent University
CS 464 Machine Learning
Final Report
Group 11

**Movie Recommendation System Using
Collaborative Filtering Neural Network with
Embeddings**

Group Members:

- | | |
|----------------------|----------|
| ● Kaan Tek | 21901946 |
| ● Kerem Şahin | 21901724 |
| ● Melih Fazıl Keskin | 21901831 |
| ● Miray Ayerdem | 21901987 |
| ● Sanaz Gheibuni | 22201383 |

Table of Contents

1. Introduction	3
1.1. Project Description	3
1.2. Structure of the Report	4
1.3. Collaborative vs Content Filtering	4
2. Data Preprocessing: Our Methodology	5
2.1. Explicit vs Implicit Feedback	5
2.2. Our Dataset	6
2.3. Train-Validation-Test Split	8
2.3.1. Train Set	9
2.3.2. Validation Set	9
2.3.3. Test Set	9
3. Coding Environment Summary	10
4. Implicit Feedback	11
4.1. Embedding	11
4.2. NCF	12
4.3. Our Implementation	13
4.4. Performance Evaluation Metric	17
4.4.1 Results of the Model	18
5. Explicit Feedback (Rating Prediction)	18
5.1. Our Implementation	19
5.2. Performance Evaluation Metric & Results of the Model	20
6. Hyperparameter Tuning	21
7. Discussion & Comparing Results of Implicit vs Explicit Rating	23
8. t-SNE	24
9. Side Notes	28
10. Workload distribution for Project & Presentation	29
10. References	29

1. Introduction

1.1. Project Description

In this project, our main goal is to build a movie recommendation system that is effective in suggesting movies to users. To achieve this, we used a technique called collaborative filtering neural network with embeddings. This technique employs neural networks to learn the similarities between users and movies based on their interactions and use this information to provide personalized recommendations.

Collaborative filtering neural network with embeddings has several advantages over traditional matrix factorization methods. For example, it can overcome the limitations of data sparsity by learning non-linear and complex user-item interaction functions.

To train our model, we used a dataset that contained unique users, movies, and ratings that users had given to the movies. We used implicit feedback in one of our models, which means that we trained our model based on the interaction between users and movies. After training this model, we evaluated its performance using the 10 Hit Ratio model in our validation set. According to the results that we obtained on this set, we tuned some of the hyperparameters of the model to improve its accuracy and performance. Then, we visualized our findings by using t-SNE and observed the clustering groups presented by t-SNE.

In our second model, we used explicit feedback, which means we trained our model based on the value of ratings given by the user and we tried to predict the rating that a user gives to a particular movie. After training this model, we evaluated its performance using the MAE performance metric in our validation set and we tuned some of the hyperparameters of our model. Finally, we evaluated its performance using MAE.

1.2. Structure of the Report

The report is structured as follows. In this first section, we give basic information about the project and provide fundamental information that creates a basis for the whole report. In Section 2, our data preprocessing methodology and our reasons for doing them is explained. In Section 3, the libraries, frameworks, and the coding environment that we have used have been summarized. In Section 4 and 5, the structure of the models has been explained in detail. In section 6, our methods for hyperparameter tuning has been explained. In section 7, the results of the models have been compared. In section 8, our application of t-SNE for the implicit feedback mechanism has been explained. In section 9, additional side notes that may make the report clearer have been explained.

1.3. Collaborative vs Content Filtering

The two methods for movie recommendation systems are content-based filtering and collaborative filtering. The way they make recommendations differs based on how they use information about users and items. Content-based filtering uses the features or properties of movies, such as genre, language, actors, etc., to identify similar movies to the ones the user has liked or rated in the past. This method does not require any data from other users during the recommendation process for a specific user. For instance, if a user has shown a preference for action movies, content-based filtering will recommend other action movies [1].

On the other hand, collaborative filtering uses the feedback or ratings of other users who have similar preferences or tastes to the target user. It does not require any domain knowledge or feature engineering for the items. It creates embeddings for both users and items independently. For example, if a user likes a movie that another user also likes, collaborative filtering will recommend other movies that the other user has also enjoyed [2].

Collaborative filtering is a recommended method that leverages the similarities between users and items to provide personalized recommendations. By analyzing the preferences of similar users, collaborative filtering models can suggest items to a user that they may not have considered before. This approach is known for generating serendipitous recommendations. In addition, the embeddings used by collaborative filtering models are learned automatically, without the need for manual feature engineering. Collaborative filtering has several advantages over content-based filtering, including the ability to recommend items that are different from what the user has liked in the past, and the absence of the need for domain knowledge or feature engineering [1].

Collaborative filtering has its own limitations and drawbacks. One of the challenges is the cold start problem. When it comes to new users or items that have no interactions or feedback, collaborative filtering cannot recommend anything. Another limitation of collaborative filtering is its inability to capture the specific interests or tastes of a user, particularly if they differ from the majority of users. Finally, collaborative filtering can also be affected by noise or bias in the user ratings or feedback. [3].

2. Data Preprocessing: Our Methodology

In this section, the methodology for preprocessing data to address the previously mentioned disadvantages will be discussed. Specifically, explicit and implicit feedback mechanisms will be covered in order to justify our reasons for transforming the data. After that, our transformation steps on the data will be explained.

2.1. Explicit vs Implicit Feedback

Explicit feedback in movie recommendation systems refers to the input given directly by viewers on the films they have seen. Users typically submit their feedback in the form of star ratings, ranging from 1 to 5, after watching a film. Machine learning models that create customized suggestions for consumers based on their preferences can be trained using explicit feedback. The recommendation engine uses this data to predict how much a user would enjoy a certain movie based on their prior ratings and reviews. The problem posed by explicit feedback movie recommendation systems, which rely on users' explicit ratings, is that users often do not provide ratings for every movie they watch. On the other hand, implicit feedback takes into account interactions between users and movies. In an implicit feedback mechanism, the system only considers whether the user watched a movie. Therefore, such a recommendation system will also be able to take into account the movies that the user has watched but not rated. This approach allows for a wider range of pattern detection, as every movie a user has interacted with can be tracked, regardless of whether they rated it or not. This implicit feedback mechanism is also used by many systems, such as Netflix and Amazon [2].

In this project, we built different models using explicit feedback and implicit feedback to witness our theoretical findings and decide which one is a better option.

2.2. Our Dataset

userId	movieId	rating	timestamp
2	5	3.0	867039249
2	25	3.0	867039168
2	32	2.0	867039166
2	58	3.0	867039325
2	64	4.0	867039612
...
270868	116797	4.0	1453408794
270868	122886	3.5	1453415583
270868	134130	4.0	1453415577
270868	134853	5.0	1453408856
270868	139385	4.5	1453415590

Figure 1: Initial form of our dataset.

We are using a dataset we found on kaggle.com [4]. This dataset has unique movie Ids, user Ids, and it has individual ratings of users from 1 to 5 for the movies that they have watched. The dataset has a structure consisting of 20M ratings, 131k unique movie IDs, and 138k unique users. On top of that, it includes timestamp information about when the movie was rated by the user and this information will be helpful for us in our validation and test split (more detail in the upcoming sections). The data can be seen in Figure 1.

For the model that uses explicit feedback, we are able to use the rating column directly since it uses the value of the ratings for training. For the model that uses implicit feedback, we need to use the user interaction information. To simulate interaction information, ratings have been converted into interactions, where each rating row is converted into a label of 1 to show that the user interacted with the movie. Expectedly, the movies that the user has not watched will have an interaction value of zero. Therefore, we will treat every rating as an interaction between the user and the movie. After this transition, our dataset will be in the following form:

userID	movieID	label
175386	1350	1
175386	156	0
175386	6872	0
175386	4634	0
175386	31221	0
...
110354	1217	1
110354	91688	0
110354	6844	0
110354	1731	0
110354	26084	0

Figure 2: Transformed form of data which captures interaction behavior.

In both models, we filtered out users with less than 20 ratings in order to recognize more meaningful user interactions, resulting in the removal of 40% of the users from the dataset. When the model is trained under ratings that are coming consistently from single users, it will be able to detect better patterns between users. Doing this would decrease the training time, reduce memory requirements, and reduce noise, with the cost of losing some information. But even with these changes, we still have data from 161.497 unique users.

```
user_ids = user_counts[user_counts > 20].index.tolist()
# create a new dataframe to use in the future and keep the original one unchanged
# this new dataframe only has the users that gave more than 20 ratings
new_X_data = pd.DataFrame({'userId': user_ids, 'count': user_counts[user_counts > 20]})
```

Code 1: Removing users with less than 20 rating

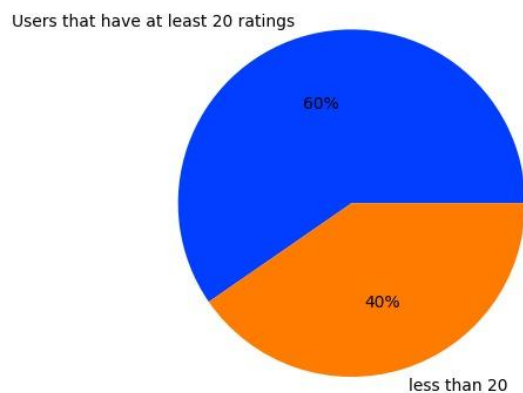


Figure 3: The proportion of users who have less than 20 ratings.

Lastly, since we have physical limitations (e.g. memory), 5% of the users were randomly selected, and their ratings were taken into account to manage memory usage. Ultimately, 1.230.738 ratings from 8074 users were used in the final data.

```
np.random.seed(123) # necessary for consistent results
# randomly select 5% of the users to simplify our data
rand_userIds = np.random.choice(new_X_data['userId'].unique(), size=int(len(new_X_data['userId'].unique())*0.05),replace=False)
# only get the rows of the selected users from the original data,
# i.e. combine the selected user ids with the rest of the information: movieId, rating, timestamp
new_X_data = X_data.loc[X_data['userId'].isin(rand_userIds)]
```

Code 2: Part of Data Preprocessing

In Figure 4, the distribution of the number of ratings each user gave can be seen. It can be seen that most of the users expectedly gave less than 1000 ratings. If the figure is carefully examined, it can be realized that the distribution is denser at ratings lower than 100.

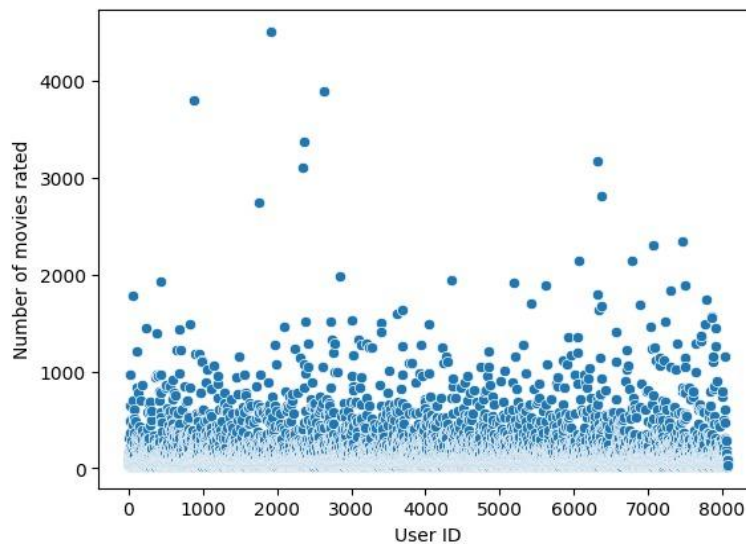


Figure 4: The distribution of the number of movies rated by users.

2.3. Train-Validation-Test Split

Our processed data is split into a train set, validation set, and test set. The newest ratings given by each user are put to the test set since predicting recent rates by looking at the previous rates is more logical. With the same logic, the second newest ratings are put into the validation set, and the remaining ratings are put into the train set.

2.3.1. Train Set

The number of ratings given by a particular user is generally less than 1000. However, we have 20.875 unique movies in the processed data. So, If we were to combine each user with each movie that exists in our dataset for the training set of the model with implicit feedback, we would get very imbalanced data since many users only interacted with a very small portion of the movies. Hence, we would have many 0 labels in our data. This would not be ideal for training the method. Therefore, in our training set, instead of taking into account every unwatched movie, we generated 4 negative samples for each row of data (positive samples). In other words, the ratio of negative to positive samples is 4:1. For the training set of the model with explicit feedback, we just included the given ratings and there was no need for additional adjustments.

2.3.2. Validation Set

The validation set will be used to determine the optimal values for the hyperparameters in the model. It contains the second newest ratings given by each user. So, it only contains positive samples. To obtain results from the validation set of implicit feedback model, we used Hit Ratio @10. For this evaluation metric, only positive samples in a set are adequate since it will combine positive samples with negative samples by itself. The details of the Hit Ratio @10 are explained in the Performance Evaluation section. To obtain results from the validation set of the explicit feedback model, we used MAE. Its details are also explained in the Performance Evaluation section.

2.3.3. Test Set

The test set is used to evaluate the performance of the final model, which is tuned during the validation stage. Again, the Hit Ratio @10 evaluation metric is used for implicit feedback models and MAE is used for explicit feedback model.

```
# rank the timestamps of ratings for each user, put them in a new column 'rank_latest'
# most recent review of that user will have the highest ranking, least recent will have the lowest
new_X_data['rank_latest'] = new_X_data.groupby(['userId'])['timestamp'].rank(method='first', ascending=False)

test_ratings = new_X_data[new_X_data['rank_latest'] == 1]
valid_ratings = new_X_data[new_X_data['rank_latest'] == 2]
train_ratings = new_X_data[new_X_data['rank_latest'] > 2]
```

Code 3: Data Splitting.

3. Coding Environment Summary

For programming the model, Python programming language is used, and we used Google Colab for running the codes. The reasons why we used Google Colab are that it is free and it allows us to work synchronously without using any version control system. Also, it offers better performance in time by allocating a GPU to run our code, and the libraries that we used were already installed in Google Colab.

The Python libraries that we used are as follows:

- Pandas & Numpy [5][6]

We used these two libraries to preprocess our dataset and make them suitable to the model that we want to build. Pandas provides a flexible and easy-to-use data structure that can handle missing data, merge and join data sets, and perform other data manipulation tasks efficiently. NumPy offers fast and efficient numerical computations and array operations, making it ideal for data processing tasks such as reshaping and transforming data. These tools enable us to quickly and effectively preprocess data, which is important for us to build our model accurately and efficiently.

- PyTorch/Lightning [7] [8]

PyTorch is a widely used library in the field of deep learning, which we chose to use for our movie recommendation model. It is known for its ease of use and flexibility, making it a popular choice for researchers and practitioners. We used the PyTorch library to build our model since it is a useful library for training deep-learning models. It also has a framework called Lightning, which provides a higher-level interface for training deep learning models. We implemented our model using both Lightning and without Lightning.

- Matplotlib [9]

Matplotlib is a popular data visualization library in Python that enables users to create different types of charts, graphs, and plots. It is a simple-to-use library that offers a number of customizable choices for developing visually appealing and informative plots. Because of these reasons, we used Matplotlib to draw necessary plots and charts, such as the distribution of the number of rated movies for each user.

- `cuml.manifold`

t-SNE is a computationally expensive method. Running t-SNE on CPU takes a lot of time, which is infeasible for our case. Therefore, `cuml.manifold`'s t-SNE implementation is used in order to run the t-SNE on GPU, instead of the CPU. In order to add the library, you first have to install the library. Then, you need to restart the runtime of the Google Collab environment. Then, you will be able to use the library in order to run t-SNE. The library reduces the runtime significantly, decreasing the runtime from hours to seconds.

4. Implicit Feedback

4.1. Embedding

The first step to implementing our model is getting familiar with the Embedding concept. An embedding layer is used to learn a low-dimensional representation of discrete inputs, such as words or categories, in continuous vector space. In a recommender system, embeddings are low-dimensional representations of high-dimensional objects like people, and products. Their objective is to concisely and meaningfully express the essence or semantics of the items. They are automatically learned from data.

The most common way to visualize embeddings is as dense vectors of floating-point values, where each dimension stands for a latent property or characteristic of the object. For instance, embedding a movie in a recommender system might have dimensions for the director, actor, and genre, among other things.

Instead of one hot encoding, which is very sparse and not useful for training a neural network, embedding can be used to learn a dense vector representation of users in our task, such that similar users are closer together in vector space.

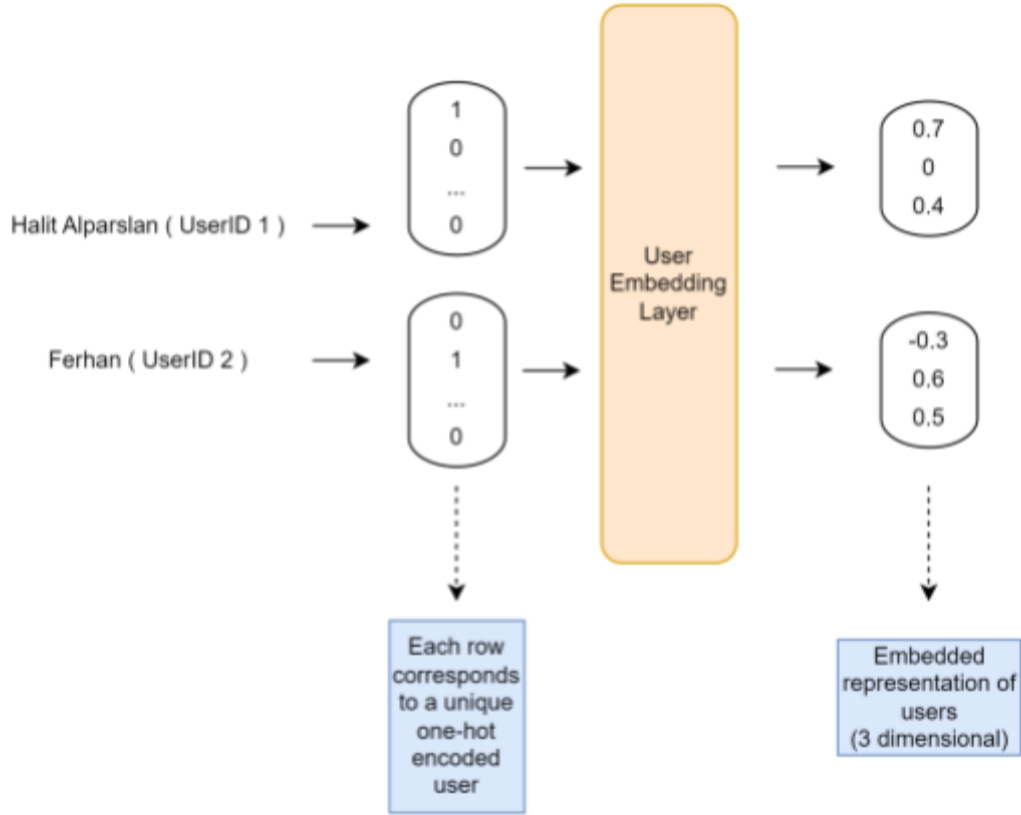


Figure 5: Example explanation of user embedding

As an example in Figure 5, when we assume that we have two users and according to their preference for three movies: the first user has more tendency to first movie than third one, and he is not interested in second movie. The second user is interested in second and last, and he has less interest in first movie.

A large number of dimensions would allow us to capture the traits of each item more accurately, at the cost of model complexity.

However, we will try different numbers of dimensions, and according to the evaluation metric, we will select an optimum embedding dimension.

4.2. NCF

Now, we will explain our Neural Collaborative Filtering model in more detail which is basically one implementation of a neural network. Hence, we are working with neurons and layers. As can be seen in the Figure 6, one instance of a user and movie are

fed to the model, and after being embedded and going through various other layers, it gives the output of the probability of the user watching that movie.

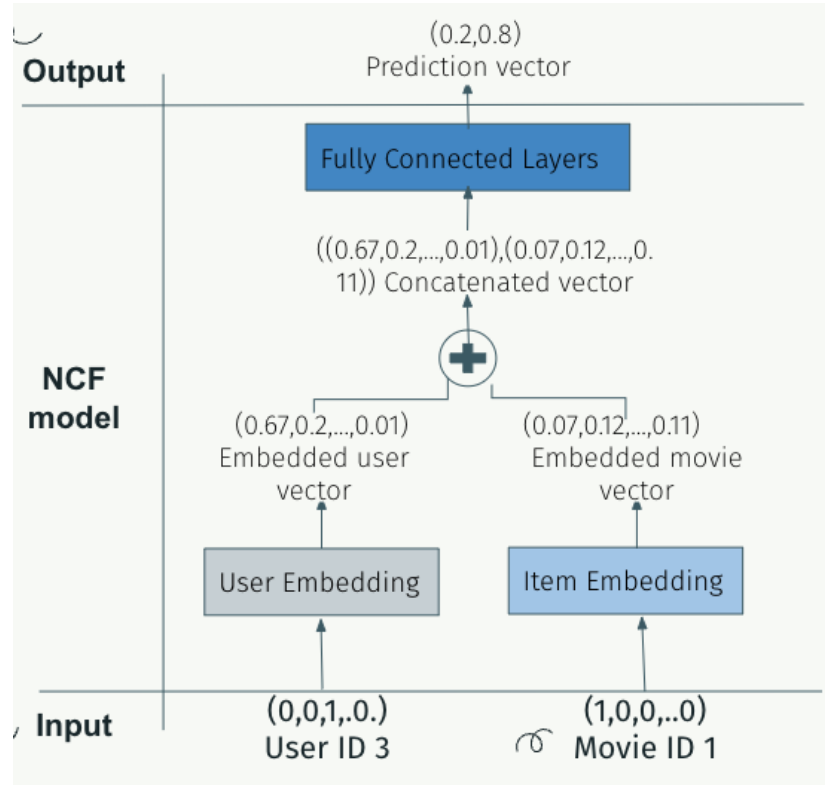


Figure 6: Example workflow of the NCF model

4.3. Our Implementation

In our first implementation of the model, we benefited from PyTorch's Lightning framework, which allows us to train models without having to write too much boilerplate code, such as for loops for epochs. In order for our model to be compatible with PyTorch's predefined methods and interfaces, we defined our model as a PyTorch dataset:

```

import torch
from torch.utils.data import Dataset

class RatingDataset(Dataset):

    # ratings: ratings data (train_ratings for training)
    # all_movie_IDs: all unique movieIDs
    def __init__(self, ratings, all_movie_IDs):
        self.users, self.items, self.labels = self.get_dataset(ratings, all_movie_IDs)

    # get length
    def __len__(self):
        return len(self.users)

    # get single instance
    def __getitem__(self, idx):
        return self.users[idx], self.items[idx], self.labels[idx]

    def get_dataset(self, ratings, all_movieIDs):
        # users: userIDs
        # items: movieIDs
        # labels: 0 or 1 : 'not interacted' or 'interacted'
        users = []
        items = []
        labels = []

        # tuple for user-item, to be able to iterate over (u, i)
        user_item_set = set(zip(ratings['userId'], ratings['movieId']))

        # 4 negative samples for each positive sample
        neg_ratio = 4
        for u, i in user_item_set:
            users.append(u)
            items.append(i)
            labels.append(1) # if it is in user_item_set, user already interacted with it
            for _ in range(neg_ratio):
                # randomly select an unwatched movie for that user
                negative_item = np.random.choice(all_movieIDs)
                # keep selecting until it is certainly not watched by that user
                while (u, negative_item) in user_item_set:
                    negative_item = np.random.choice(all_movieIDs)
                users.append(u)
                items.append(negative_item)
                labels.append(0) # add 0, because the user has not watched that movie

        # convert lists to PyTorch tensors
        return torch.tensor(users), torch.tensor(items), torch.tensor(labels)

```

Code 4: Pytorch Dataset

In our neural network model, we have 5 layers in total (selected by hyperparameter tuning, see Section 6). 2 of them are used to embed user and movie IDs, respectively. Then these embeddings are concatenated together and fed into three other dense, in other words, fully connected layers. These three layers use a ReLU function. The ReLU function is used to introduce non-linearity to our model. Finally, we arrive at the output layer, which consists of a single neuron that has a Sigmoid activation function. This neuron gives an output value between 0 and 1, which represents the probability that the user will like the item. This result is then used as a ranking score to recommend the top 10 items to the user, which is explained more in detail later. In the first fully

connected layer, we have 64 neurons, in the second layer, we have 32 neurons. Finally, the last layer has one neuron, which gives us the final predicted output.

In order to implement embeddings, layers, and activation functions; we used Torch's nn module, as it comes with pre-defined building blocks. Our model uses Binary Cross Entropy as its loss function, which calculates the loss of our model with every iteration and then we use Adam Optimizer -which is one type of stochastic gradient descent- to update our weights and biases. This process of forward pass of training and loss calculation, and then the backpropagation of weight adjusting continues until our model converges or the specified number of epochs is reached.

```
import torch.nn as nn
import pytorch_lightning as pl
from torch.utils.data import DataLoader


class ModelWithLightning(pl.LightningModule):

    # num_users: total number of unique users
    # num_items: total number of unique movies
    # ratings: rating data (train_ratings for training)
    # all_movieIds: all unique movie ids in all data
    # embd_dimension: embedding dimension
    def __init__(self, num_users, num_items, ratings, all_movie_IDS, embd_dimension, layers, lr, batch):
        super().__init__()
        # init embedding layers
        self.layers = layers
        self.lr = lr
        self.batch_size = batch
        self.user_embedding = nn.Embedding(num_embeddings=num_users, embedding_dim=embd_dimension)
        self.item_embedding = nn.Embedding(num_embeddings=num_items, embedding_dim=embd_dimension)
        # init fully connected and output layer
        if layers == 2:
            self.dense1 = nn.Linear(in_features=16, out_features=64)
            self.dense2 = nn.Linear(in_features=64, out_features=32)
            # init output layer
            self.output = nn.Linear(in_features=32, out_features=1)
        elif layers == 3:
            self.dense1 = nn.Linear(in_features=16, out_features=64)
            self.dense2 = nn.Linear(in_features=64, out_features=32)
            self.dense3 = nn.Linear(in_features=32, out_features=16)
            # init output layer
            self.output = nn.Linear(in_features=16, out_features=1)
        self.ratings = ratings
        self.all_movie_IDS = all_movie_IDS
        self.training_step_outputs = []
```

Code 5: Model Using Lightning

```

num_users = new_X_data['userId'].max()+1
num_items = new_X_data['movieId'].max()+1
all_movie_ids = new_X_data['movieId'].unique()
embd_dimension = 8
model = ModelLightning(num_users, num_items, train_ratings, all_movie_ids, embd_dimension)
trainer = pl.Trainer(max_epochs=7, reload_dataloaders_every_n_epochs=True, logger=False)
trainer.fit(model)

INFO:pytorch_lightning.utilities.rank_zero:GPU available: False, used: False
INFO:pytorch_lightning.utilities.rank_zero:TPU available: False, using: 0 TPU cores
INFO:pytorch_lightning.utilities.rank_zero:IPU available: False, using: 0 IPUs
INFO:pytorch_lightning.utilities.rank_zero:HPU available: False, using: 0 HPUs
INFO:pytorch_lightning.callbacks.model_summary:
| Name | Type | Params |
|-----|-----|-----|
0 | user_embedding | Embedding | 2.2 M
1 | item_embedding | Embedding | 1.4 M
2 | fc1 | Linear | 1.1 K
3 | fc2 | Linear | 2.1 K
4 | fc3 | Linear | 528
5 | output | Linear | 17
|-----|-----|-----|
3.6 M Trainable params
0 Non-trainable params
3.6 M Total params
14.322 Total estimated model params size (MB)
Epoch 6: 100%  12050/12050 [13:48<00:00, 14.54it/s]
INFO:pytorch_lightning.utilities.rank_zero:`Trainer.fit` stopped: `max_epochs=7` reached.

```

Code 6: Training of the model with Lightning using Trainer class

4.4. Performance Evaluation Metric

In order to evaluate the performance of our model, we had to choose a performance metric. Traditional performance metrics such as accuracy, precision, and F1-score are not always suitable for recommendation systems because the traditional performance metrics evaluate the performance of a model based on its ability to predict individual labels, rather than to rank a large number of items based on their relevance. Additionally, recommendation systems usually deal with highly imbalanced datasets, where the majority of items have no interaction or relevance to the user, and only a small portion of items are actually relevant. This makes it challenging to evaluate the performance of a recommendation system using traditional metrics, which may be biased toward the majority class. Hence, we used another metric: Hit Ratio @ 10.

Hit Ratio @ 10 is an evaluation metric commonly used in recommender systems [10] to measure the ability of a model to recommend relevant items to a user. It is defined as the percentage of cases where at least one of the top 10 recommended items matches a held-out test item that the user actually interacted with. In other words, if a user has interacted with any of the top 10 recommended items, the recommendation is considered a "hit". This makes sense because imagine your streaming service recommends you 10

movies, you see the list, and even if you watch only one movie from the list, you could say that the streaming service did a good job with its recommendations.

In order to implement this logic in our model, we made some additions to our validation set. Our initial validation set contained one instance for each user, which has the rating that the user gave to their most recently watched movie, along with the user and movie id. We added randomly chosen 99 unwatched movies for each user to our validation set, and combining it with the watched item that already exists in the set, we had a total of 100 items for each user (same logic applies to the test set as well). Then for each user, we feed our model with these 100 items, and in the end, our model outputs the probability of the user watching that item for all those 100 items. We rank those items based on their returned probabilities and choose the top 10 items with the highest probabilities among all. Then we check whether the initial watched item is in the top 10 recommendations, if it is, it is a hit, else, it is a miss. Then we calculate the average of the hits, and it is our final hit ratio.

4.4.1 Results of the Model

Once we trained our model with our training set and tuned our hyperparameters on the validation set we evaluated our final model on our test dataset with 2 hidden layers, 512 batch size, and 0.01 learning rate. As a result, we obtained an 86.9% percent hit ratio, which can be interpreted as in 86.9% of the recommendation lists that our model generates, the user watches one of the movies from the list.

```
[ ] print("10 Hit Ratio for the best model on test data: ", end="")
    final_hit_ratio = evaluate(best_model)

10 Hit Ratio for the best model on test data: The Hit Ratio @ 10 is 0.869086
```

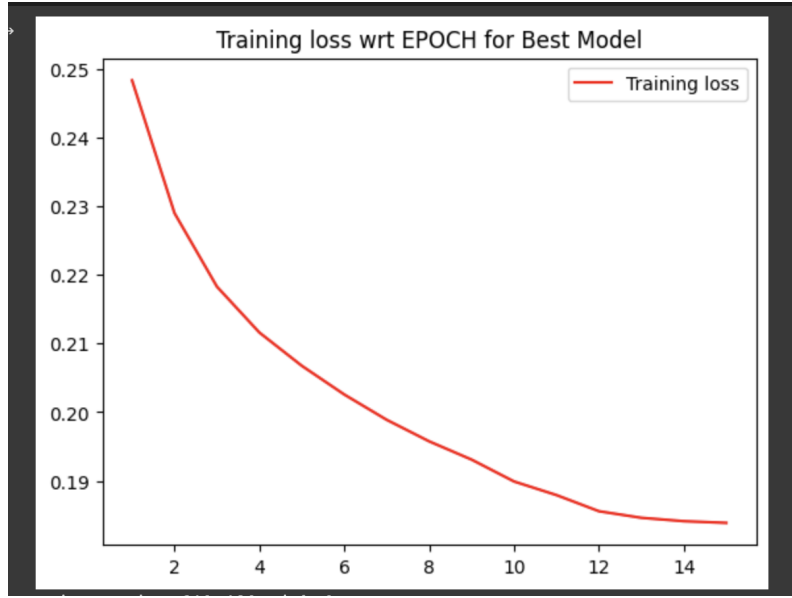


Figure 7: Training of the Implicit model with Lightning

5. Explicit Feedback (Rating Prediction)

Until this point, we used implicit ratings to train our model. In our model, recommendations are based on user interaction, and consequently, the model ignores the rating information. Explicit rating information can also be incorporated to predict how much the user will like a particular movie. Using explicit feedback for movie rating prediction can be beneficial because it provides a direct measure of the user's preference. The model can use this information to estimate the rating that a user would assign to an unseen item. The model uses similar principles to the model for implicit feedback in terms of embedding layers and NCF logic. However, this time we have edited the loss functions, relevant embeddings, and the output function in order to train our model as a model which gives predictions.

5.1. Our Implementation

```
class ModelWithLightningDOT(pl.LightningModule):

    def __init__(self, num_users, num_items, ratings, all_movie_IDs, embd_dimension, layers, lr, batch):
        super().__init__()
        self.layers = layers
        self.lr = lr
        self.batch_size = batch
        self.user_embedding = nn.Embedding(num_embeddings=num_users, embedding_dim=embd_dimension)
        self.item_embedding = nn.Embedding(num_embeddings=num_items, embedding_dim=embd_dimension)
        if layers == 2:
            self.dense1 = nn.Linear(in_features=16, out_features=64)
            self.dense2 = nn.Linear(in_features=64, out_features=32)
            self.output = nn.Linear(in_features=32, out_features=1)
        elif layers == 3:
            self.dense1 = nn.Linear(in_features=16, out_features=64)
            self.dense2 = nn.Linear(in_features=64, out_features=32)
            self.dense3 = nn.Linear(in_features=32, out_features=16)
            self.output = nn.Linear(in_features=16, out_features=1)

        self.training_step_outputs = []
        self.ratings = ratings
        self.all_movie_IDs = all_movie_IDs

    def forward(self, user_input, item_input):
        user_embedded = self.user_embedding(user_input)
        item_embedded = self.item_embedding(item_input)

        tensor = torch.cat([user_embedded, item_embedded], dim=-1)

        tensor = nn.ReLU()(self.dense1(tensor))
        tensor = nn.ReLU()(self.dense2(tensor))
        if self.layers == 3:
            tensor = nn.ReLU()(self.dense3(tensor))

        pred = nn.Sigmoid()(self.output(tensor))
        pred = (pred * 4) + 1 # Scale the predictions to the range [1, 5]

        return pred
```

Figure 8: Rating Prediction Model

Figure 8 shows our implemented model for rating prediction. The model has two embedding layers: one for users and one for movies. These layers map the user and item to a continuous vector space. The predicted rating is obtained by passing the concatenated user and item embeddings through the fully connected layer and scaling the output to range [1,5] by multiplying it by four and adding 1. This is because the ratings in the dataset are assumed to be in this range. This model uses 3 hidden layers, as it was the best performing for our hyperparameter tuning (See hyperparameter tuning section) In the first fully connected layer, we have 64 neurons, in the second layer, we have 32 neurons, and the third layer has 16 neurons. Finally, the last layer has one neuron, which gives us the final predicted output. After tuning parameters, we train the model on the train dataset with learning rate of 0.001 and batch size of 256.

For the loss function, we use MSE. According to Figure 9 , in each epoch model tries to capture the pattern, and overall loss decreases:

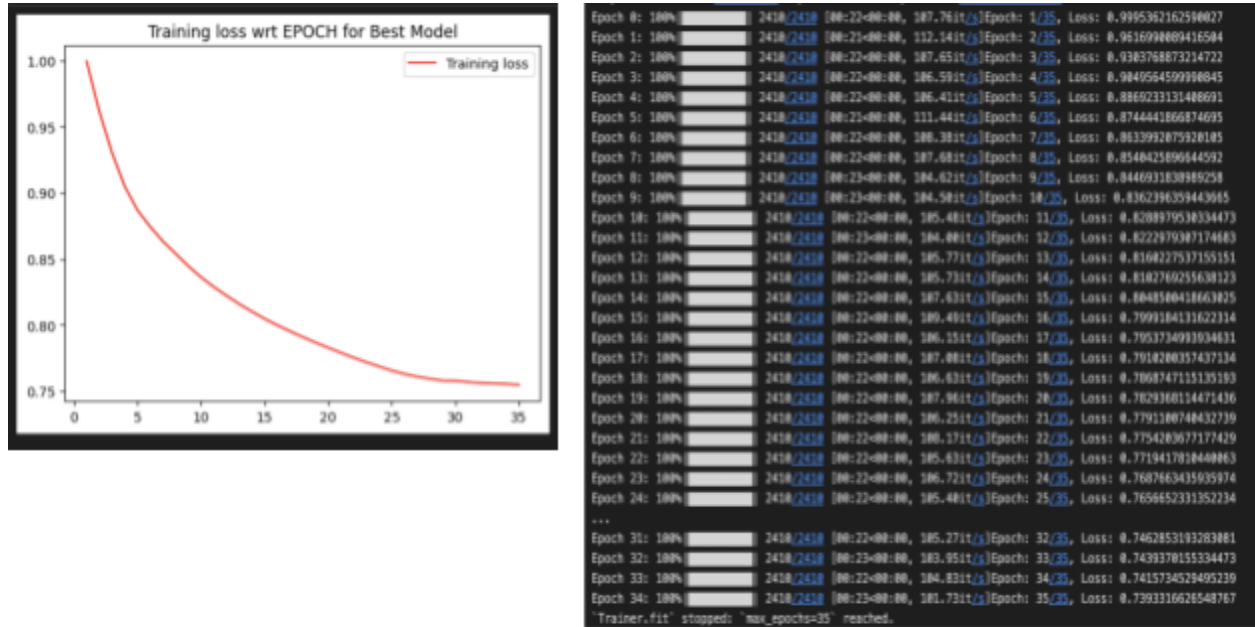


Figure 9: Training Loss

5.2. Performance Evaluation Metric & Results of the Model

In the previous model, we were using the Hit Ratio @10 method. However, for this model, it is not an appropriate method to assess our model because Hit Ratio @10 measures the ability of the model to recommend movies that the user would rate within the top 10. While Hit Ratio @10 captures the notion of recommendation success, it does not directly measure the accuracy of predicted ratings. However, we aim to measure the accurate rating predictions in our model. Therefore, we will be using another metric. Simply, we will be looking at its Mean Absolute Error (MAE) because MAE directly measures the absolute difference between predicted and actual ratings, providing a quantitative assessment of the model's accuracy. It evaluates the overall prediction performance across all ratings. In its essence, model is acting as a linear regression model which predicts continuous values between [1,5] (They are mapped to integers while giving prediction as results). This means that MAE is applicable as an evaluation metric for this model.

```
from sklearn.metrics import mean_absolute_error
mean_absolute_error(test_ratings['rating'], rounded_pred)
✓ 0.0s
0.7626331434233342
```

Figure 10: Model Evaluation

According to Figure 10, MAE value came out to be 0.76. We can interpret this value as follows. This means that when the model is predicting ratings, the ratings it give are off by ± 0.76 . This is a relatively good result because in real life, it is a complicated task to predict the exact rating that a user would give to a movie.

6. Hyperparameter Tuning

Hyperparameter tuning is done in order to determine the best combination of hyperparameters and come up with the optimal model. The models and the results that have been shown until here are all hyperparameter tuned. Now, we will be explaining how these results were gathered by also explaining our hyperparameter procedure.

Our hyperparameters were the *size of our network*, *batch size*, and *learning rate*. We tested two combinations as the size of the model, one having 2 hidden layers with 64 and 32 neurons in each, and another having 3 hidden layers with 64, 32 and 16 neurons in each. We also tested two combinations of batch sizes, 256 and 512. The batch sizes we used were relatively huge, but we used them because after our initial tests with smaller batch sizes, we experienced that the training took a lot of time due to our large data, and the time required for convergence was not that much different than larger batch sizes, so after our initial experiments, we decided to continue our experiments with larger batch sizes. For learning rate, we tested three different values being 0.001, 0.01, and 0.1. Therefore, in total, we trained a total of 12 models with our training data, and evaluated them on our validation data.

6.1. Implicit Feedback Tuning

```
[ ] num_users = new_X_data['userId'].max()+1
    num_items = new_X_data['movieId'].max()+1
    all_movie_ids = new_X_data['movieId'].unique()
    embd_dimension = 8

    # Hyperparameters to be tuned
    hyperparams = {
        "layers" : [2, 3],
        "batch_sizes" : [256, 512],
        "alphas" : [0.001, 0.01, 0.1]
    }

    # Model With Lightning

    all_models = []
    for layer in hyperparams["layers"]:
        for batch_size in hyperparams["batch_sizes"]:
            for alpha in hyperparams["alphas"]:
                print(f"Model with layers: {'[16, 64]' if layer == 2 else '[16, 64, 32]'}, batch_size: {batch_size}, lr: {alpha}")
                model_lightning = ModelWithLightning(num_users, num_items, train_ratings, all_movie_ids, embd_dimension, layer, alpha, batch_size)
                trainer = pl.Trainer(max_epochs=7, logger=False)
                trainer.fit(model_lightning)
                all_models.append(model_lightning)
```

Code 7: Hyperparameter tuning

Once the models were trained, we evaluated them on our validation set by using our Hit Ratio @10 method. As can be seen in the code 7, two models gave 1 out of 1 hit ratio, but we suspected that there could be some kind of overfitting during the training of these models, and in fact, there was, because when we then tested them on our test data, the hit ratio significantly dropped. So we decided to choose among the next two best-performing models, which have around a 0.88 hit ratio. The only difference between them was the size of the network, so we decided to use the model with 2 layers as it is less complex and performs the same as the more complex model with 3 layers. Hence, our final model has 2 hidden layers, 512 batch size, and 0.01 learning rate.

```
[ ] alpha_idx = -1
    history = []
    for i, model in enumerate(all_models):
        alpha_idx += 1
        print(f"Model with layers: {'[16, 64]' if i < 6 else '[16, 64, 32]'}, ", end="")
        print(f"batch_size: {hyperparams['batch_sizes'][0 if i < 3 or (i >= 6 and i <= 8) else 1]}, ", end="")
        print(f"learning_rate: {hyperparams['alphas'][alpha_idx]}: ", end="")
        if (i+1) % 3 == 0: alpha_idx = -1
        history.append(evaluate(model))

Model with layers: [16, 64], batch_size: 256, learning_rate: 0.001: The Hit Ratio @ 10 is 0.860788
Model with layers: [16, 64], batch_size: 256, learning_rate: 0.01: The Hit Ratio @ 10 is 0.548179
Model with layers: [16, 64], batch_size: 256, learning_rate: 0.1: The Hit Ratio @ 10 is 1.000000
Model with layers: [16, 64], batch_size: 512, learning_rate: 0.001: The Hit Ratio @ 10 is 0.810131
Model with layers: [16, 64], batch_size: 512, learning_rate: 0.01: The Hit Ratio @ 10 is 0.886178
Model with layers: [16, 64], batch_size: 512, learning_rate: 0.1: The Hit Ratio @ 10 is 0.816324
Model with layers: [16, 64, 32], batch_size: 256, learning_rate: 0.001: The Hit Ratio @ 10 is 0.854100
Model with layers: [16, 64, 32], batch_size: 256, learning_rate: 0.01: The Hit Ratio @ 10 is 0.878870
Model with layers: [16, 64, 32], batch_size: 256, learning_rate: 0.1: The Hit Ratio @ 10 is 0.759475
Model with layers: [16, 64, 32], batch_size: 512, learning_rate: 0.001: The Hit Ratio @ 10 is 0.833044
Model with layers: [16, 64, 32], batch_size: 512, learning_rate: 0.01: The Hit Ratio @ 10 is 0.889027
Model with layers: [16, 64, 32], batch_size: 512, learning_rate: 0.1: The Hit Ratio @ 10 is 1.000000
```

Code 8: Evaluation of the models trained with different hyperparameters

We then trained a new model with the final chosen parameters by using our training data, and tested it on the test data. The plot you see is the training loss of the model at the end of each iteration, and the final hit ratio we obtained was 0.87 .

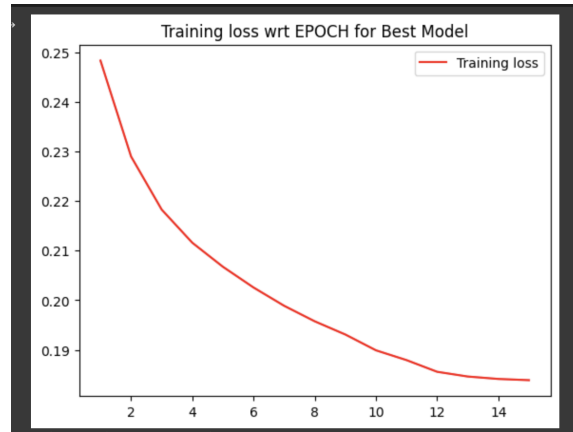


Figure 11: Training loss for the final model

```
print("10 Hit Ratio for the best model on test data: ", end="")
final_hit_ratio = evaluate(best_model)

10 Hit Ratio for the best model on test data: The Hit Ratio @ 10 is 0.876146
```

+ Code

Code 9: Evaluation of the final model

6.1. Explicit Feedback Tuning

```
num_users = new_X_data['userId'].max() + 1
num_items = new_X_data['movieId'].max() + 1
all_movie_IDs = new_X_data['movieId'].unique()
embd_dimension = 8

# Hyperparameters to be tuned
hyperparams = {
    "layers": [2, 3],
    "batch_sizes": [256, 512],
    "alphas": [0.001, 0.01, 0.1]
}

all_models = []
for layer in hyperparams["layers"]:
    for batch_size in hyperparams["batch_sizes"]:
        for alpha in hyperparams["alphas"]:
            print(f"Model with layers: {'[16, 64]' if layer == 2 else '[16, 64, 32]'}, batch_size: {batch_size}, lr: {alpha}")
            model_lightning = ModelWithLightningDOT(num_users, num_items, train_ratings, all_movie_IDs, embd_dimension, layer, alpha, batch_size)
            trainer = pl.Trainer(accelerator='cpu', max_epochs=15, logger=False)
            trainer.fit(model_lightning)
            all_models.append(model_lightning)
```

```

Model with layers: [16, 64], batch_size: 256, learning_rate: 0.001: MAE error is: 0.8319296507307407
Model with layers: [16, 64], batch_size: 256, learning_rate: 0.01: MAE error is: 0.886177854842705
Model with layers: [16, 64], batch_size: 256, learning_rate: 0.1: MAE error is: 0.8910081743869209
Model with layers: [16, 64], batch_size: 512, learning_rate: 0.001: MAE error is: 0.815518949715135
Model with layers: [16, 64], batch_size: 512, learning_rate: 0.01: MAE error is: 0.852365618033193
Model with layers: [16, 64], batch_size: 512, learning_rate: 0.1: MAE error is: 0.8910081743869209
Model with layers: [16, 64, 32], batch_size: 256, learning_rate: 0.001: MAE error is: 0.8881496160515234
Model with layers: [16, 64, 32], batch_size: 256, learning_rate: 0.01: MAE error is: 0.8098835769135496
Model with layers: [16, 64, 32], batch_size: 256, learning_rate: 0.1: MAE error is: 0.8910081743869209
Model with layers: [16, 64, 32], batch_size: 512, learning_rate: 0.001: MAE error is: 0.8174386920980926
Model with layers: [16, 64, 32], batch_size: 512, learning_rate: 0.01: MAE error is: 0.8794897200891751
Model with layers: [16, 64, 32], batch_size: 512, learning_rate: 0.1: MAE error is: 0.8910081743869209

BEST MODEL:
Model with layers: [16, 64, 32], batch_size: 256, learning_rate: 0.001: MAE error is: 0.8881496160515234 (Due to memory purposes, we could not train 35 epochs which is the epoch number for training)

```

Once the models were trained, we evaluated them on our validation set by using our MAE method. Different from the implicit method, the 3 layered version gave the best results. As a side note, we used 15 epochs due to the time constraints. However, after our parameter tuning, we have used 35 epochs which is why the hyperparameter tuned versions MAE was higher than the MAE value we observed on the test set. If we were able to use 35 epochs, our MAE value would probably be less. However, this is not critical for this section because we are comparing each combination on 15 epochs. Model with 3 layers, batch_size: 256, learning_rate: 0.001 gave the best result and it has MAE error: 0.8081496160515234. We then trained on our dataset and gathered the results (which were already shown in Section 5)

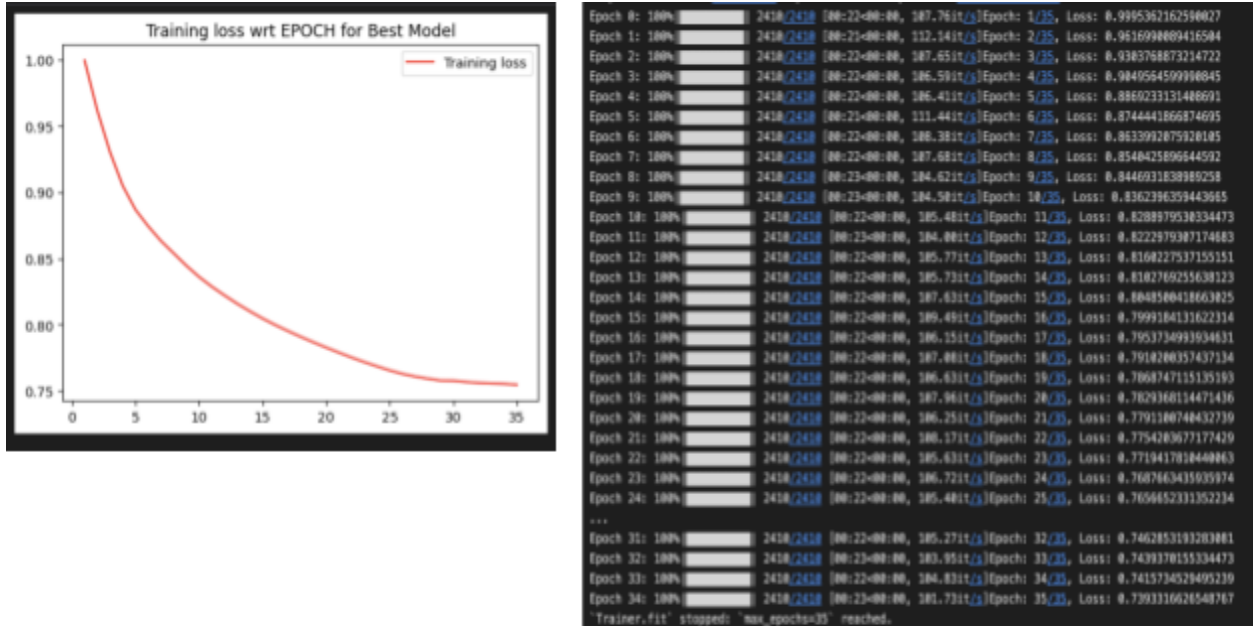


Figure 12 : Training Loss

7. Discussion & Comparing Results of Implicit vs Explicit Rating

Explicit feedback can be sparse as not all users may provide ratings or reviews. This sparsity can limit the effectiveness of explicit feedback-based recommendation

systems, especially in scenarios where a large portion of the user-item interactions are missing. On the other hand, implicit feedback is more readily available, as users leave traces of their preferences through their actions. This makes implicit feedback a valuable resource for making recommendations, especially when explicit feedback is scarce. Hence, implicit feedback mechanism is used much more in the industry as it provides a more feasible and a realistic approach to giving recommendations to users, instead of trying to predict their exact rating. The following table contains results and a comparison of implemented two models:

Methods	Loss Function	Evaluation Metric	Result on Validation dataset	Result on Test dataset
Implicit	BCE	Hit Ratio @10	0.88	0.869
Explicit	MSE	MAE	0.808	0.76

As mentioned before, we used different loss functions, and evaluation metrics for our two models because the idea behind their implementation is slightly different. The first model tries to recommend movies that the user is likely to watch. On the other hand, the second model tries to predict the rating that a potential user would give to a movie. According to our research and results, we believe that both have given accurate results in terms of the context that they are going to use.

The first model has received a 0.87 hit percent. This means that the model was successful in predicting which movies that the user is going to watch. This makes the model sensible to be used in the industry because the main idea is to make people watch more movies. According to the first model, roughly 9 out of 10 times, the recommended movie would be watched.

For the second model, it was able to predict the ratings that a user would give to a movie with an offset of 0.76. This means that the model is able to give a good prediction on what rating a user would give to a movie.

Even if both are giving decent results in terms of their own contexts, we would decide to move on with the implicit rating. First of all, the main objective of using this model within the perspective of a company is to make people watch more movies. Therefore, recommending movies that a user likes does not necessarily guarantee a high

rate of watching a movie. Secondly, the implicit feedback mechanism is more widely used in the industry. Therefore, there is more evidence for the usefulness of the first model and it would be our selection for a real-life setting.

8. t-SNE

t-SNE is a dimensionality reduction technique used for visualizing high-dimensional data in a lower-dimensional space. It preserves local relationships between data points by mapping similar points in the original space to nearby points in the lower-dimensional space. For this case, we have used t-SNE to visualize the similarities in our embeddings. The resulting representation can be visualized to reveal clusters in the data. We have used this technique to verify that our model could create embeddings that found similarities. We will apply t-SNE to our implicit model, as all the other models also use the same embedding logic.

Before we applied t-SNE, certain data preprocessing was needed. In our code, we need to use the maximum ID value while defining the matrix sizes. This is needed as the indexing makes use of the values of actual ID's. However, this situation results in a very high dimensionality. If one of the ID's is very high when compared to the actual number of movies (Max Movie ID: 30.000, Num of Movies: 8000), the code unnecessarily creates more cells than it needs to. Therefore, a mapping where each movie ID was set to the number of movies in a sequential manner has been applied. This reduces the size of our embedding matrices, as indexing is done according to the ids.

```
unique_movie_ids = list(set(new_X_data['movieId'])) # Get unique movie IDs
unique_movie_ids.sort() # Sort the unique IDs in ascending order

# Create a dictionary to map the original movie IDs to sequential numbers
movie_id_mapping = {id: i+1 for i, id in enumerate(unique_movie_ids)}

# Map the movie IDs in your dataset using the created dictionary
new_X_data['movieId'] = [movie_id_mapping[id] for id in new_X_data['movieId']]

unique_user_ids = list(set(new_X_data['userId'])) # Get unique movie IDs
unique_user_ids.sort() # Sort the unique IDs in ascending order

# Create a dictionary to map the original movie IDs to sequential numbers
user_id_mapping = {id: i+1 for i, id in enumerate(unique_user_ids)}

# Map the movie IDs in your dataset using the created dictionary
new_X_data['userId'] = [user_id_mapping[id] for id in new_X_data['userId']]

test_ratings = new_X_data[new_X_data['rank_latest'] == 1]
valid_ratings = new_X_data[new_X_data['rank_latest'] == 2]
train_ratings = new_X_data[new_X_data['rank_latest'] > 2]
```

Figure 13: ID Mapping for Movies and Users

Other than that, we had to apply parameter tuning to t-SNE parameters in order to observe the clustering. The main parameters that are required to be adjusted are the perplexity, `n_iter`, and `n_neighbours`. The perplexity and `n_neighbours` parameters control a similar thing, which is the number of points it considers while the model tries to find clusterings. The `n_iter` is trivial, but it is important as t-SNE is not a very fast algorithm. Therefore the `n_iter` parameter is important to optimize to get a decent clustering and a decent runtime. As a side note, the number of epochs that our model is trained on also affects the clustering because as we increase the number of epochs, the model is able to find better similarities. In order to run the code faster, we have found a different t-SNE implementation which uses GPU instead of CPU. The selected values for parameters can be seen as follows in Figure 14:

```
num_users = 8075
# Combine user and item embeddings into a single matrix
combined_embeddings = np.concatenate((user_embeddings, item_embeddings), axis=0)

# Apply t-SNE to obtain lower-dimensional representations
tsne = TSNE(n_components = 2, random_state = 42, n_iter=10000, perplexity=80 ,n_neighbors=240)
embeddings_2d = tsne.fit_transform(combined_embeddings)

# Separate user and item embeddings
user_embeddings_2d = embeddings_2d[:num_users]
item_embeddings_2d = embeddings_2d[num_users:]

# Apply t-SNE to obtain lower-dimensional representations
tsne = TSNE(n_components = 2, random_state = 42, n_iter=325)
user_embeddings_2d = tsne.fit_transform(user_embeddings)

tsne2 = TSNE(n_components = 2, random_state = 42, n_iter=325)
item_embeddings_2d = tsne2.fit_transform(item_embeddings)
```

Figure 14: Configuration of t-SNE

The t-SNE has been applied to two different embeddings. First, the concatenated embedding which is used as the input to Fully Connected Layers has been given to t-SNE. The embedding is a concatenation of the user and movie embedding layers. Visualizing this will provide information on whether the model was able to detect certain similarities between users and movies. When we apply the t-SNE, we get the following

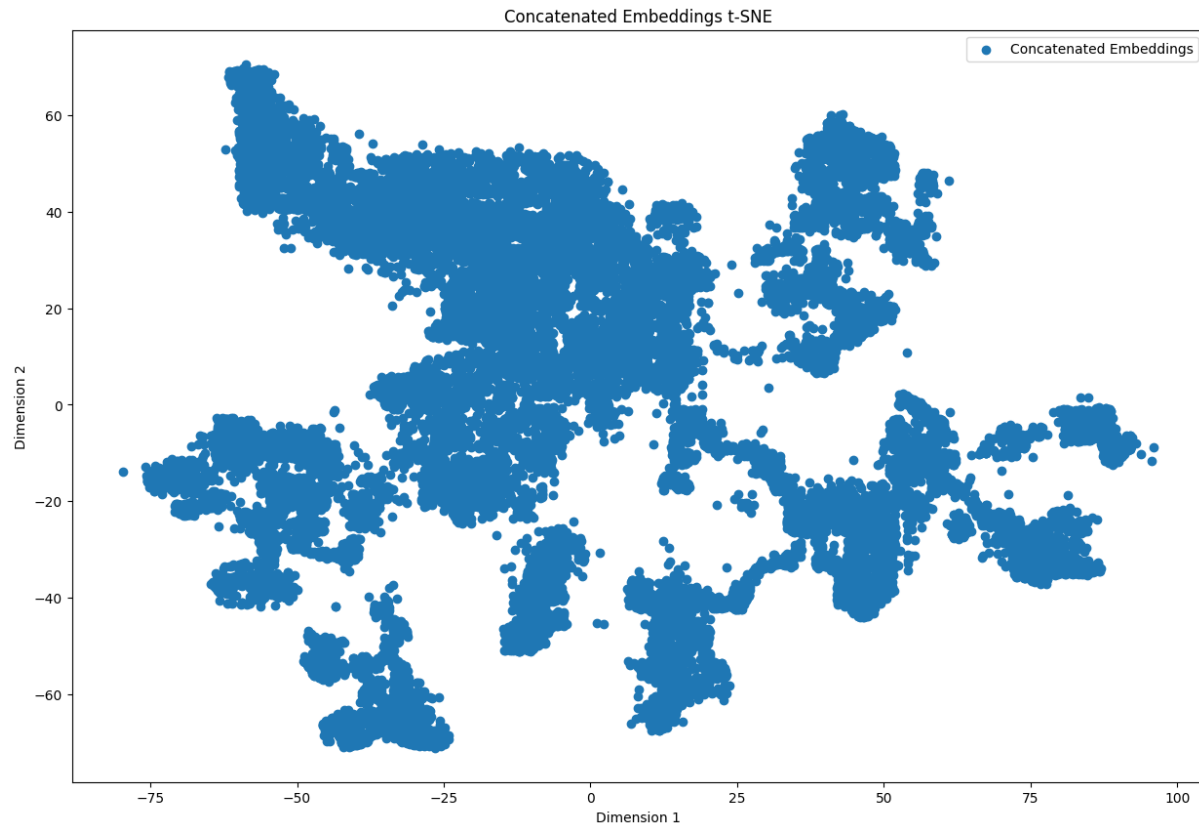


Figure 15: t-SNE applied on concatenated embeddings

As we can see, there are dense clusters which represent close data points that our embedding found. Some of the clusters are not well defined. However, this can be expected for any recommendation system because it is not possible to find exact similarities with each user and movie, which will result in a not so perfect clustering. Considering this information, the clusterings of our embeddings can be considered as a success. Intuitively, when our model is recommending, it will logically recommend movies to users according to these similarities.

On top of this concatenated embedding, we also wanted to visualize our movie embedding in order to see whether our model was able to relate certain movies with each other. When we apply the same procedure on our movie embedding, we get the following result (Figure 16):

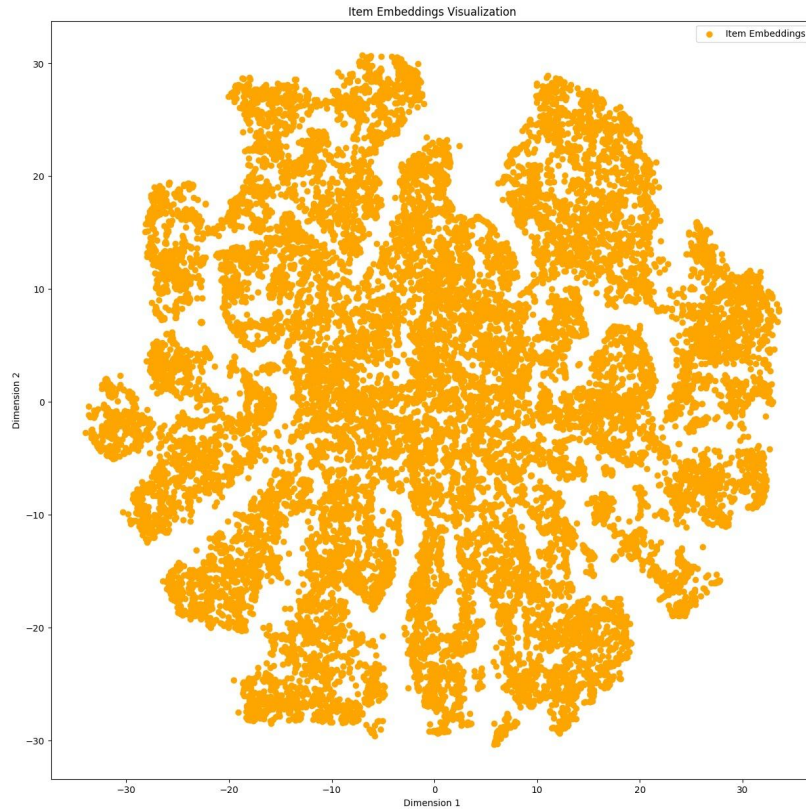


Figure 16: t-SNE applied on movie embeddings

As it can be seen in Figure 16, certain clusters are formed which indicates that our model was able to relate certain movies with each other. The clusters that are surrounding the circle in the middle are well defined, which means that our model was able to learn certain similarities between these movie data points. In conclusion, t-SNE can be seen as an informal proof that the model is working as expected, and it finds certain similarities with different data points.

9. Side Notes

In the previous report, we implemented an implicit rating without lightning. However, after our hyperparameter tuning and analysis, we saw that the model with lightning gave results faster and it was more feasible and easier to maintain. Therefore, we did not include it in this project. On top of that, we have added a new model, which is the explicit feedback mechanism. We have added hyperparameter tuning and t-SNE as well. Lastly, we have edited the previous sections to be consistent with our latest results.

10. Workload distribution for Project & Presentation

Most of the work was done together.

Kaan Tek: Data Preprocessing Part of Code | Progress Report | Helped implementation of Neural Collaborative Filtering Model with Lightning | NCF part of Presentation | Implementation of Hyperparameter Tuning | Helped implementation of Explicit Rating Model | Tuning Part of Final Report

Kerem Şahin: Data Preprocessing Section of Presentation | Progress Report | Helped implementation of Embedding Model | Data Preprocessing Part of Code | t-SNE implementation | Final Report

Melih Fazıl Keskin: Data Preprocessing Part of Code | Train Test Split Section of Progression Presentation | Progress Report | Helped implementation of Train Test Split | Helped implementation of Explicit Feedback model | Helped Dataset Processing for t-SNE | Final Report | Revision Part of Final Presentation

Miray Ayerdem: Data Preprocessing Section of Code | Helped implementation of Embedding Model | “Information about Recommendation Systems” Section of Presentation | Helped implementation of Explicit Feedback model | Implicit Model Implementation Part of Final Presentation | Final Report

Sanaz Gheibuni: Data Preprocessing Part of Code | Progress Report | Helped implementation of Neural Collaborative Filtering Model with Pytorch | Trained the Model | Helped implementation of Explicit Feedback model | Embedding and Movie rating prediction (Explicit Feedback) part of Presentation | Final Report

10. References

- [1] “Collaborative filtering | machine learning | google developers,” *Google*. [Online]. Available: <https://developers.google.com/machine-learning/recommendation/collaborative/basics>. [Accessed: 23-Apr-2023].
- [2] “Introduction to Recommender Systems,” *Medium*. [Online]. Available: <https://towardsdatascience.com/deep-learning-based-recommender-systems-3d120201db7e>. [Accessed: 24-Apr-2023].
- [3] “Collaborative Filtering Advantages & disadvantages | machine learning | google developers,” *Google*. [Online]. Available: <https://developers.google.com/machine-learning/recommendation/collaborative/summary>. [Accessed: 27-Apr-2023].
- [4] GroupLens, “MovieLens 20M Dataset,” *Kaggle*, 15-Aug-2018. [Online]. Available: <https://www.kaggle.com/datasets/groupLens/movielens-20m-dataset>. [Accessed: 27-Apr-2023].

- [5] “Pandas,” *pandas*. [Online]. Available: <https://pandas.pydata.org/>. [Accessed: 27-Apr-2023].
- [6] *NumPy*. [Online]. Available: <https://numpy.org/>. [Accessed: 27-Apr-2023].
- [7] “PYTORCH documentation,” *PyTorch documentation - PyTorch 2.0 documentation*. [Online]. Available: <https://pytorch.org/docs/stable/index.html>. [Accessed: 27-Apr-2023].
- [8] “Pytorch Lightning,” *Lightning Open Source*. [Online]. Available: <https://www.pytorchlightning.ai/index.html>. [Accessed: 27-Apr-2023].
- [9] “Matplotlib 3.7.1 documentation#,” *Matplotlib documentation - Matplotlib 3.7.1 documentation*. [Online]. Available: <https://matplotlib.org/stable/index.html>. [Accessed: 27-Apr-2023].
- [10] Kane, Frank. Building Recommender Systems with Machine Learning and AI. Packt Publishing, 2018.