PMLDL assignment 2

Adelina Kildeeva B21-DS-02

a.kildeeva@innopolis.university

Movie Recommender System

Final Report

Introduction

A recommender system is a type of information filtering system that suggests items or content to users based on their interests, preferences, or past behavior. In this assignment I used MovieLens-100k dataset to create a movie recommender system.

Data analysis

To create a model I used 3 files from the dataset: u.data, u.item, and u.user.

The u.data consists of the user_id, movie_id and rating from 1 to 5. Figure 1 shows the distributions of the ratings. Each user has rated from 20 to 737 movies. Most of the ratings vary from 3 to 5.

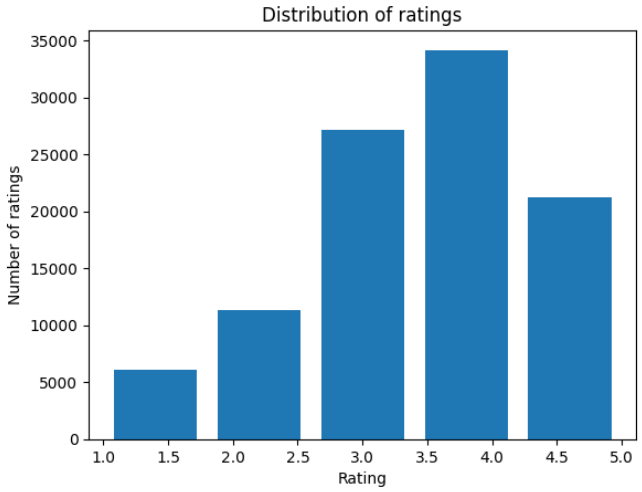

Figure 1

The u.item contains the movie information (title, release date, url, and genre). Video release date is null for all movies. Therefore, I dropped this column. Genres represented in a convenient way and used for model training.

The u.user includes the user metadata (age, gender, occupation, zipcode). Figures 2 and 3 show distributions of users age and occupation. Gender and occupation were transformed for model trainings (using get_dummies). Zipcode is dropped and is not used for the model.

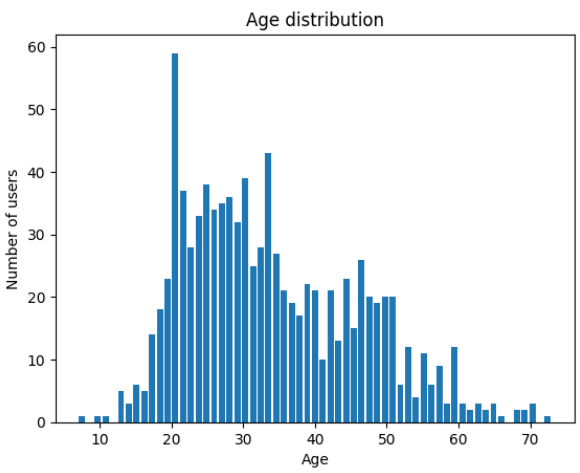

Figure 2

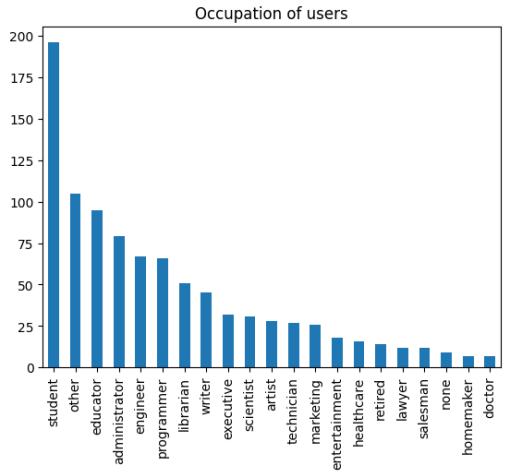

Figure 3

Data exploration shows that dataset is mostly cleaned and can be used to train a model. For the recommendation system training I will use ratings, movie genre, and user metadata (age, gender, and occupation). More detailed analysis and data preprocessing can be found in notebooks/1.0-initial-data-exploration.ipynb.

Model Implementation

For recommendation system I chose a collaborative filtering approach with an embedding layer for user and movie ids, and a fully connected neural network for additional features like genre, age, gender, and occupation. Figure 4 shows the architecture of the model.

```python
class MovieRecommendationModel(nn.Module):
    def __init__(self, num_users, num_movies, num_features, embedding_dim=50, hidden_dim=100):
        super(MovieRecommendationModel, self).__init__()

        # Embedding layers for user and movie IDs
        self.user_embedding = nn.Embedding(num_users, embedding_dim)
        self.movie_embedding = nn.Embedding(num_movies, embedding_dim)

        # Fully connected layers for additional features
        self.fc = nn.Sequential(
            nn.Linear(embedding_dim * 2 + num_features, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, 1)
        )

    def forward(self, user, movie, features):
        # Embed user and movie IDs
        user_embedding = self.user_embedding(user)
        movie_embedding = self.movie_embedding(movie)

        # Concatenate user and movie embeddings with additional features
        input_features = torch.cat([user_embedding, movie_embedding, features], dim=1)

        # Forward pass through fully connected layers
        output = self.fc(input_features)

        return output.squeeze()
```

Figure 4. Model Architecture

Model Advantages and Disadvantages

1. Advantages

- Personalization. The model leverages embeddings for users and movies, enabling personalized recommendations based on their preferences and behaviors. It captures user-item interactions, enhancing the recommendation quality.
- Considering additional features. By incorporating additional user or movie features (such as genres, age, gender, etc.) into the model, it can potentially improve recommendations by considering diverse factors.
- Flexibility and customization. The architecture's modularity allows for easy customization and expansion. It is possible to adjust embedding dimensions, hidden layer sizes, or add more features to enhance performance.

2. Disadvantages

- Data Sparsity. If users or movies have limited interactions or feature information, the model's ability to make accurate recommendations might be limited.
- Cold start problem. New users or movies might not have enough interactions or data to provide meaningful recommendations. But in the given dataset each user and movie have enough ratings to make meaningful predictions.
- Overfitting. Deeper architectures are prone to overfitting if not properly regularized or validated. Balancing model complexity with generalization to prevent overfitting is crucial.

Training process

The model is trained on MSE loss with learning rate 0.01 on 50 epochs. The best model is saved to the models/model.pt and used further for evaluations. Figure 5 represents a chart of train/validation loss of the model over epochs.

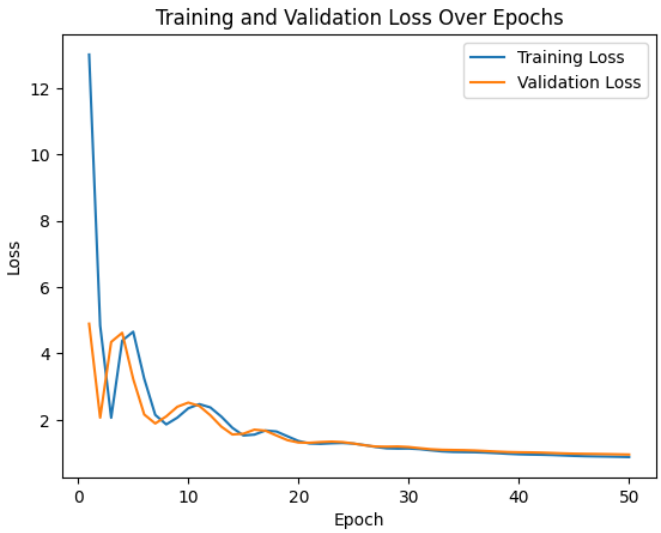

Figure 5

Evaluation

[1] contains detailed overview of the key metrics for recommender systems. After analyzing the page, I decided to choose rating metrics RMSE and MAE.

RMSE is for evaluating the accuracy of prediction on ratings. RMSE is the most widely used metric to evaluate a recommendation algorithm that predicts missing ratings. MAE computes the metric value from ground truths and prediction in the same scale. Compared to RMSE, MAE is more explainable.

The results of evaluation.py: RMSE = 0.9834, MAE = 0.7804.

Results

I wrote a collaborative filtering model with a fully-connected neural networks for additional features using MovieLens-100k dataset. The model was trained on MSE loss on 50 epochs. The best model is saved to the ‘models’ folder. The evaluate.py script allows to evaluate the performance of the model using RMSE and MAE metrics. The results of the implemented model: RMSE = 0.9834, MAE = 0.7804. The system shows good results for the given dataset.

References

[1] Overview of key metrics for recommender systems https://github.com/recommenders-team/recommenders/blob/main/examples/03_evaluate/evaluation.ipynb