PMLDL Assignment 2 Final Report

Movie Recommender System with Neural Networks

Lev Rekhlov, BS21-DS-02, l.rekhlov@innopolis.university

1. Idea.

To create a recommender system with the provided dataset(users, movies and ratings), I started from the ground: how it will work? based on what data? what it will output?

Here are several variants I came up with:

- **User + Movie -> Rating**. In such setup, the system should get information about user, then the model predicts "what rating will user give to each film in database" and outputs top K suggestions.
- **Movie + Ratings -> Users**. Based on known ratings, the model outputs who will like this movie and recommends it them.
- **User + Ratings -> Movies**. We 'capture' the user behavior and trying to predict movies which user is likely to watch next.

As a rule of thumb, I want my recommender system to be simple and easily repeatable. My decision was: **User + Movie -> Rating**.

Such system can be easily solved as **classification task**: model predicts one of 5 ratings, and then we sort all result by rating, giving only top 5-10 to user.

2. Feature Engineering and dataset preparation.

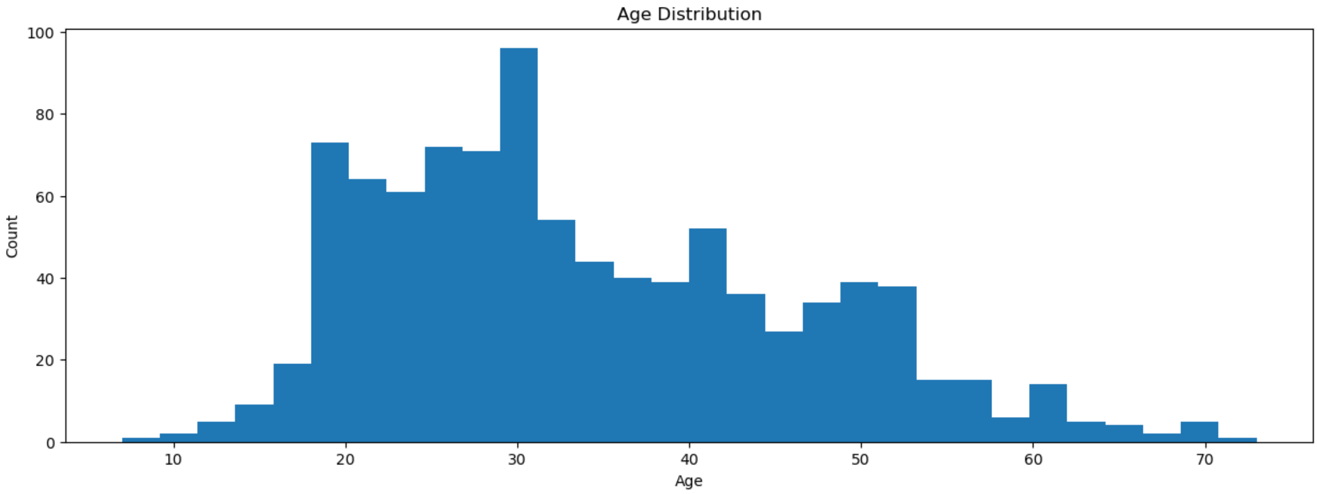
Let's create features for each type of entities: User and Movie, step by step.

2.1 User Representation

Initial dataset gives us the following knowledge about user: `gender` , `age` , `occupation` and `zipcode` .

`Gender` and `occupation` columns can be easily encoded using **One-Hot Encoding**. It simple, yet efficient method to encode categorical variables.

The `age` column contains integers from 7 to 73. Let's look at histogram:



The number of different ages is high, so I decided to categorize them into bins:

- 0-18 age
- 18-25 age
- 25-35 age
- ...

In total, I got 6 bins, which are also One-Hot Encoded.

2.2 Movie Representation

As for the movies: we have `title` , `release_date` and `genres` columns. Other columns I preliminarily filtered out, as well as the rows with 'unknown' genre.

Column `genre` was again One-Hot Encoded(wow).

For the `release_date` I used the same strategy as for the `age` column. Firstly, I left only the year of release, as most of the movies were released at 1'st of January. Then, I divided the years into bins and One-Hot Encoded them.

The most interesting part is the `title` column. First idea that came to me was to encode the title using **BERT**. Why:

- Old but gold. BERT is a well-known model, which is used in many NLP tasks.
- Easy to do. Many articles/repos/tutorials on the Internet.
- Pretrained BERT already can encode words very well without fine-tuning.

So, I used it and as a result got 768-dimensional vector for each movie title, which is a sum of embeddings for each word in title.

2.3 Final Prepared Dataset

After calculating all necessary features, I merged them with `ratings` dataset. Resulting columns:

5 rows × 826 columns pd.Dataframe

	user_id	item_id	rating	timestamp	genderF	genderM	administrator	artist	doctor	educator	engineer	entertainment	executive	healthcare	homemaker	lawyer
0	196	242	3	881250949	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	186	302	3	891717742	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
2	22	377	1	878887116	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	244	51	2	880606923	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	166	346	1	886397596	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0

- `user_id` and `movie_id` - identifiers of user and movie
- `timestamp` - timestamp of rating(will be used later for train/test split)
- `rating` - rating given by user to movie
- `bert0...bert767` - 768-dimensional vector of movie title
- 18 columns of movie genres
- 6 columns of user age
- 7 columns of movie release year
- 21 columns of user occupation

This dataset can be used for training/testing in the following way: given user + movie embedding -> predict rating.

3. Baseline Model

To me, it was obvious which model I want to try first: **Decision Tree**. It's features:

- Clearly understandable features importance
- Interpretability
- Fast Training

So, I split the whole dataset into train and test parts using stratification by ratings. By this I ensured representativeness of both splits.

Without any setup and using default parameters, sklearn DecisionTree classifier gives the following results on test set:

```
              precision    recall  f1-score   support

           1       0.51      0.12      0.20      1222
           2       0.26      0.03      0.05      2274
           3       0.35      0.38      0.36      5428
           4       0.39      0.64      0.48      6834
           5       0.43      0.25      0.32      4240

    accuracy                           0.38     19998
   macro avg       0.39      0.28      0.28     19998
weighted avg       0.38      0.38      0.35     19998
```

Some analysis:

- As dataset has more 3 and 4 ratings than others, the model performs better on them.
- The best precision 0.51 is for rating 1 (least popular)
- The best recall 0.64 is for rating 4 (most popular)
- The worst precision and recall is for rating 2 (hardest to learn?)

Now, we can try to find some better model architecture!

4. Final Model

4.1 Model Architecture

For the final model, I decided to use **Linear Model**. The architecture is the following:

- Firstly, Movie Title embedding is passed through **nn.BatchNorm1d(768)** layer
- Then it goes to separate **nn.Linear(768, 128)** layer with **nn.ReLU** activation
- The output of the layer is concatenated with other features
- The concatenated vector is passed through **nn.Linear(128 + 29 + 25, 1)** layer

4.2 Train and Test Split

To be able to correctly evaluate the model, I used the following strategy:

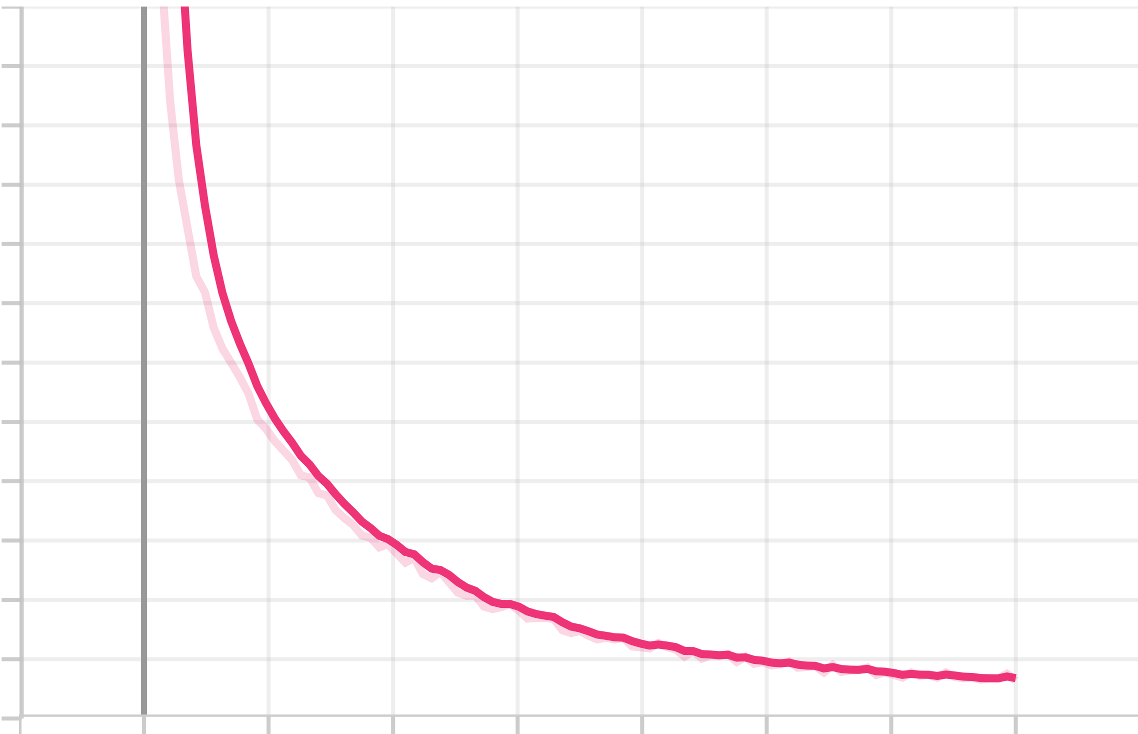
- Both train and test sets contain all users.
- All user ratings are **sorted by timestamp**.
- First 80% of ratings are used for training, last 20% for testing.

The idea is that we want to predict the rating of the movie that the user has not yet rated. **Given the past -> predict the future.**

4.3 Training Process

The model was trained using **MSE loss** and **Adam optimizer**. The learning rate was set to 0.0005.

The number of epochs was set to 100. Here is the loss plot(**RMSE** is used for better visualization):



It seems that the model can be trained for more epochs, but after some experiments, I decided to stop at 100.

5. Evaluation

First of all, I evaluated the model on the test set. I got **1.05 RMSE**. Using evaluate.py script, you can reproduce the same result.

6. Conclusion

Provided solution performs not so well, but it was interesting experiment. There exists a lot of ways for building recommendation systems.

I tried to build one using **Decision Tree** and **Linear Model**. The first one is easy to understand and interpret, and the second one is fast to train and can be easily scaled.