Machine Learning Final Project - Recommendation System

Falaras Apostolis
2981
afalaras@uth.gr

Maipas Kosmas
2567
kmaipas@uth.gr

Sialtsis Alexandros
3083
asialtsis@uth.gr

January 2024

Abstract

Due to the rising popularity of services like Netflix, Recommendations Systems are more used and important than ever. That's why this report outlines the development and evaluation of a movie Recommendation System. The project's objectives were to test the predictive capabilities of machine learning models in a movie dataset that contains just movie ratings and their titles.

First of all, we conducted exploratory data analysis on the ratings in order to capture patterns in terms of the way users review movies as well as to gain insights into the overall popularity of movies. Subsequently, we executed a variety of machine learning models and matrix factorization techniques to test each method's performance using the mean squared error metric.

1 Introduction

Recommendation systems, also commonly referred to as recommender systems, are software applications that use a subset of machine learning methods and algorithms. Their goal is to analyze user preferences about content and products, in order to make personalized recommendations to the users about content and products they haven't interacted with and might find interesting. There are various types of recommendation systems, including collaborative filtering, content-based filtering, and hybrid methods. Each type consists of it's distinct methods and algorithms.

Recommendation systems play a crucial role in today's digital world, as they can be found across various diverse applications. They are responsible for helping users navigate and find what they are looking for amid the overwhelming amount of information on the internet. Additionally, they can adapt to the users' evolving preferences, ensuring that users stay engaged to the application as their interests change over time. This not only leads to a more pleasant and time-efficient user experience but also it increases business profit as businesses can retain more customers.

The algorithm that we built takes an integer (1-1000) as input, which represents a user in our data set (Netflix Prize Dataset). Then, for that specific user, we recommend the top 10 movies for him to watch, based on his preferences. The algorithm that we use for that is the SVD algorithm, which turned out to be the second best performing algorithm for our dataset, while also being simple and straight-forward.

2 Literature Review

- **An Improved Approach for Movie Recommendation System:** The authors, Shreya Agrawal and Pooja Jain, proposed a hybrid recommendation system that combines content-based and collaborative filtering techniques. The newly introduced approach utilizes a Support Vector Machines (SVM) classifier to discover latent factors that determine the users preferences and then employs a genetic algorithm that optimizes the combination of collaborative and content-based techniques. The authors used three datasets to test the scalability of their model and they concluded that it improved the accuracy and quality of the system[1].

- **Personalized Real-Time Movie Recommendation System: Practical Prototype and Evaluation:** The authors, Jiang Zhang, Yufeng Wang, Zhiyuan Yuan, and Qun Jin, present a system that integrates collaborative and content-based filtering, as well as Virtual Opinion Leader (VOL) techniques. Specifically, VOL techniques exploit the opinions of influential users and they can provide recommendations that deviate from the popular choices. That way these techniques offer a wide variety of suggestions. The authors concluded that their system could leverage the strengths of each individual technique, and achieve high accuracy and user satisfaction.[2]

- **Neural Collaborative Filtering:** The authors, Xiangnan He, Lizi Liao and others introduced a novel architecture for collaborative filtering called Neural Collaborative Filtering (NCF). NCF is a deep learning model that learns embeddings for users and items, and then combines these embeddings to predict user ratings. The paper showed that NCF performs better than traditional methods for recommending items on different datasets. It also found new areas for research in recommendation systems, like applying deep learning to tasks beyond collaborative filtering, such as content-based and hybrid recommendations. The paper is highly praised, seen as a big contribution to the field, and has played a role in advancing recommending systems with deep learning techniques.[3]

- **Matrix Factorization Techniques for Recommender Systems** (by Yehuda Koren, Robert Bell, and Chris Volinsky (2009)): This paper discusses the winning solution to the Netflix Prize, where the authors employed matrix factorization techniques to improve collaborative filtering. The strength of their approach lies in its ability to handle the sparsity of the Netflix dataset and effectively model user preferences. The authors used a combination of singular value decomposition and stochastic gradient descent to factorize the user-item interaction matrix. One limitation is that the method doesn't explicitly consider temporal dynamics in user preferences. Despite this, the paper provides valuable insights into the application of matrix factorization to large-scale recommendation problems, and it played a crucial role in advancing the field.[4]

- **A Survey of Collaborative Filtering-Based Recommender Systems: From Traditional Methods to Hybrid Methods Based on Social Networks** (Rui Chen, Qingyi Hua, Yan-Shuo Chang, Bo Wang, Lei Zhang, and Xiangjie Kong (2018)): This paper serves as a comprehensive survey of collaborative filtering methods, providing an in-depth exploration of the various techniques employed in recommendation systems. The survey covers a wide range of collaborative filtering approaches, including memory-based and model-based methods, neighborhood-based techniques, matrix factorization, and hybrid models. It focuses into the strengths and weaknesses of each approach, outlining the challenges and advancements in collaborative filtering research up to 2011. This survey

is a valuable resource for researchers and practitioners seeking a holistic understanding of collaborative filtering methods and their evolution over the years.[5]

- **Product Recommendation Based on Content-based Filtering Using**: The authors Zeinab Shahbazi and Yung-Cheol Byun explore the use of XGBoost, an ensemble learning algorithm, for product recommendations. The paper proposes a framework that combines product attributes with user preferences to make personalized recommendations. The authors demonstrate that XGBoost can effectively capture complex relationships between product attributes and user preferences, leading to improved recommendation accuracy and even outperforming other traditional methods.[6]

3 Dataset & Features

We decided to work with the Netflix Prize Dataset. It provides a vast amount of user ratings and interactions with movies which is crucial for the accuracy of the system. It is also an interesting dataset as the data derived from actual user behavior on the Netflix platform, reflecting real-world usage patterns and preferences. The dataset can be found at:https://www.kaggle.com/datasets/netflix-inc/netflix-prize-data

The dataset contains four text files (combined_data_1.txt, combined_data_2.txt, combined_data_3.txt and combined_data_4.txt) and a csv file (movie_titles.csv). Each one of the text files contains the ratings for the movies in the following format:

```
MovieId:
CustomerID_1,Rating_1,Date_1
CustomerID_2,Rating_2,Date_2
```

For example:

```
1:
1488844,3,2005-09-06
822109,5,2005-05-13
2:
2059652,4,2005-09-05
1666394,3,2005-04-19
```

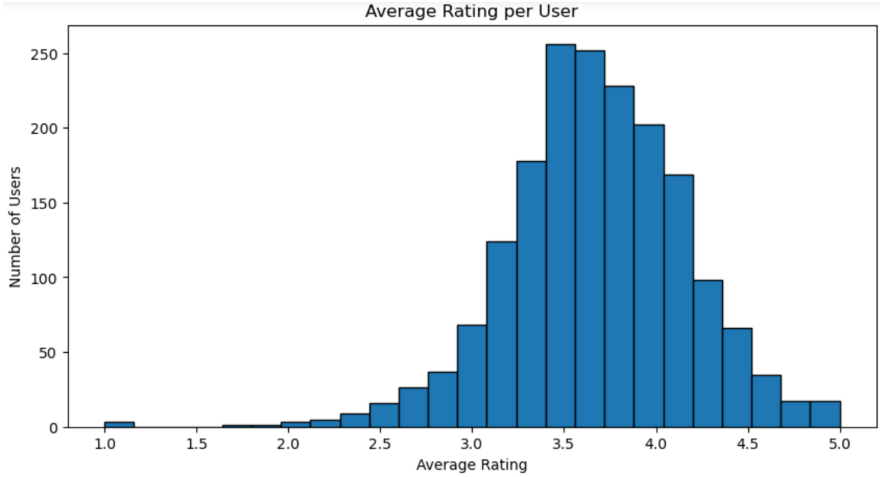
The movie_titles.csv file contains the title and release year for each movie.

The first step in the process of building a recommendation system is to perform Exploratory Data Analysis (EDA) on the input dataset. Before we can do that we need to reorganize the input dataset into a format suitable for EDA. Specifically, we decided to create four CSV files, one for each input text files, and then import then into four separate dataframes, which then we concatenate into one unified dataframe.

The first step of EDA is to inspect the dataframe structure, check for missing and duplicate values and find total number of users and movies. Next, the most important step of EDA is to investigate the ratings of the dataframe. Specifically, we plot the ratings distribution and and compute their descriptive statistics.

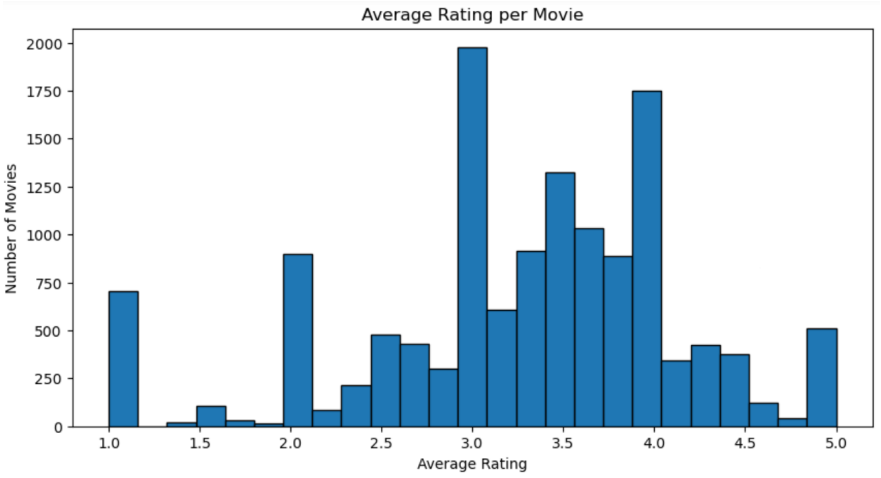
Also, we performed an analysis on the per-user ratings and per-movie ratings o gain valuable insights into user behavior and movie preferences within the recommendation system. Analysis on the per-user ratings can help us understand how users engage with the system and gain insights on user behaviour and preferences in the recommendation system. We can identify the most active users by calculating the number of ratings per user. Moreover, we can examine the

average rating of users, in order to determine whether or not users have been satisfied with the recommendation system, which is crucial feedback on the system's effectiveness.



From the average rating-per-user histogram, we observe a left-skewed normal distribution of ratings with a center of about 3.5, which suggests that users are getting recommendations including both their movies preferences and those they may not prefer, offering a diverse set of suggestions.

Turning our attention to the analysis on the per-movie ratings that can help us obtain useful information about user preferences as well as distinct movie characteristics. By calculating both the number of ratings and the average rating per movie, we can not only identify blockbuster movies with high popularity, but also discover "hidden-gems" - movies with very high average rating and low popularity.



From the average rating-per-movie histogram, we observe that the distribution doesn't resemble the normal distribution. The average rating of all movies was 3,27.

4 Machine Learning Methods

4.1 Singular Value Decomposition (SVD)

Singular Value Decomposition (SVD) is a matrix factorization technique used in collaborative filtering, that discovers the latent factors by identifying patters in the user-item interactions. It decomposes the original user-item ratings matrix into three matrices: $A = U\Sigma V^T$

- A_{mxn} : The original user-item ratings matrix.
- U_{mxk} : The left singular vectors matrix. U represents latent factors for user, which capture underlying characteristics or preferences of users. Each row of U corresponds to a user, and the values in that row indicate how much the user aligns with each latent factor.
- Σ_{kxk} : The diagonal matrix containing the singular values, which represent the importance of each latent factor.
- V_{nxk} : The right singular vectors matrix. The columns of V^T represent latent factors for the items. Each row of V^T corresponds to an item and the values in that row indicate how much the item exhibits each latent factor.

4.2 Non-Negative Matrix Factorization (NMF)

Non-Negative Matrix Factorization (NMF) is a group of algorithms in multivariate analysis and linear algebra where a matrix V is decomposed into two matrices: $V = W \cdot H$

- V_{mxn} : The original user-item ratings matrix.
- W_{mxk} : Non-negative matrix representing the user profiles in terms of the latent factors. Users with similar interaction patters have similar values in certain columns.
- H_{kxn} : Non-negative matrix representing the item profiles in terms of the latent factors. Items that are associated with similar latent factors will have similar values in certain columns in H.

4.3 Stochastic Gradient Descent (SGD)

The Stochastic Gradient Descent (SGD) algorithm aims to learn latent factors for users and items in a recommendation system. The process involves randomly initializing latent factor matrices for users (U) and items (V), which represent hidden features contributing to user-item interactions. Through a series of iterations, the algorithm stochastically selects and processes data points, calculating the prediction error by comparing the actual rating with the predicted rating, which is the dot product of user and item latent factors. The model's performance is evaluated using the Mean Squared Error (MSE) between actual and predicted ratings.

4.4 K Nearest Neighbors (KNN)

With the K-nearest neighbors (KNN) algorithm, the goal is to predict a user's rating for a particular item (movie) by considering the ratings of users who are most similar to them. After splitting our dataset into training and test sets, we can use the Nearest Neighbors algorithm to find the K users whose preferences are most similar to a given user's preferences. For each user-movie pair in the test set, the algorithm identifies the K nearest neighbors from the training

set and calculates the predicted rating as the mean of these neighbors' ratings. This is very computationally expensive. That is why we choose to use the "kd_tree" algorithm. KD-trees are more efficient than brute-force search algorithms, especially for large datasets. The Mean Squared Error (MSE) is then computed to evaluate how well the model's predictions match with the actual ratings in the test set.

4.5 XGBoost

XGBoost, which stands for eXtreme Gradient Boosting, is a powerful and efficient open-source implementation of the gradient boosting framework. It was developed by Tianqi Chen and is widely used for machine learning tasks, particularly in structured/tabular data and is known for its speed and performance.

XGBoost is based on the gradient boosting framework, which is an ensemble learning method. It builds a series of weak learners (usually decision trees) sequentially, where each tree corrects the errors made by the previous ones. It incorporates L1 (LASSO) and L2 (Ridge) regularization terms in its objective function. This helps prevent overfitting by penalizing overly complex models. Moreover, XGBoost uses a technique called "pruning" during the tree-building process. Trees are grown deep and then pruned backward, removing branches that do not provide significant contributions to reducing the loss function.

4.6 XGBoost + Baseline

A baseline model is essentially a simple model that acts as a reference in a machine learning project. Its main function is to contextualize the results of trained models. Baseline models usually lack complexity and may have little predictive power. In this project, we are going to take a baseline model from the Surprise library that uses the SGD algorithm.

Combining this Baseline model with XGBoost will help the latter gain an extra perspective on the dataset because the baseline models capture aspects of the data that the XGBoost model might struggle with or overlook. XGBoost will use the Baseline model's predictions to help predict on its own.

4.7 Neural Collaborative Filtering

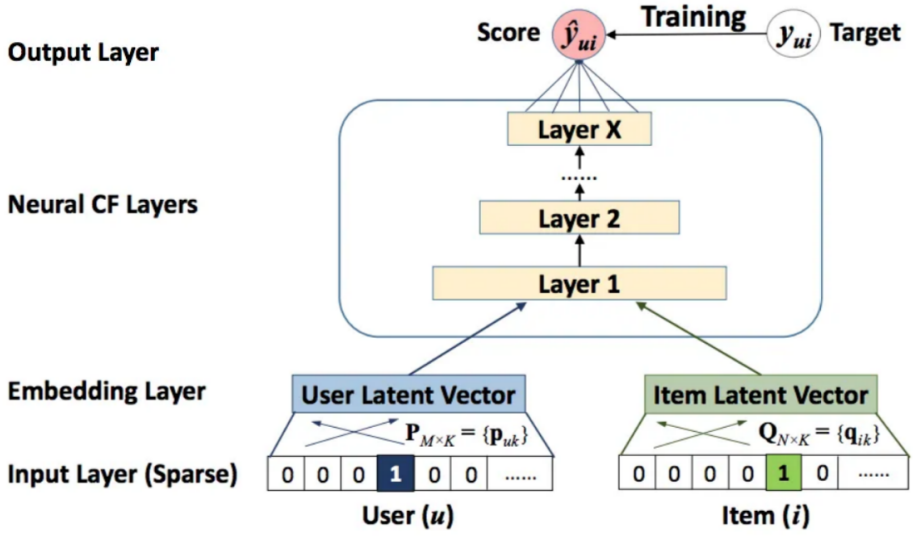
Neural Collaborative Filtering (NCF) is a type of recommendation system that leverages neural networks to model user-item interactions. NCF was introduced to address some of the limitations of traditional collaborative filtering methods, such as matrix factorization, by incorporating non-linearities and capturing complex patterns in user-item relationships.

Neural Collaborative Filtering(NCF) replaces the user-item inner product with a neural architecture. NCF tries to express and generalize MF under its framework. NCF tries to learn user-item interactions through a multi-layer perceptron.

Intuitively speaking the recommendation algorithms estimates the scores of unobserved entries in Y, which are used for ranking the items. Formally speaking they calculate:

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

The architecture we are going to use is this:



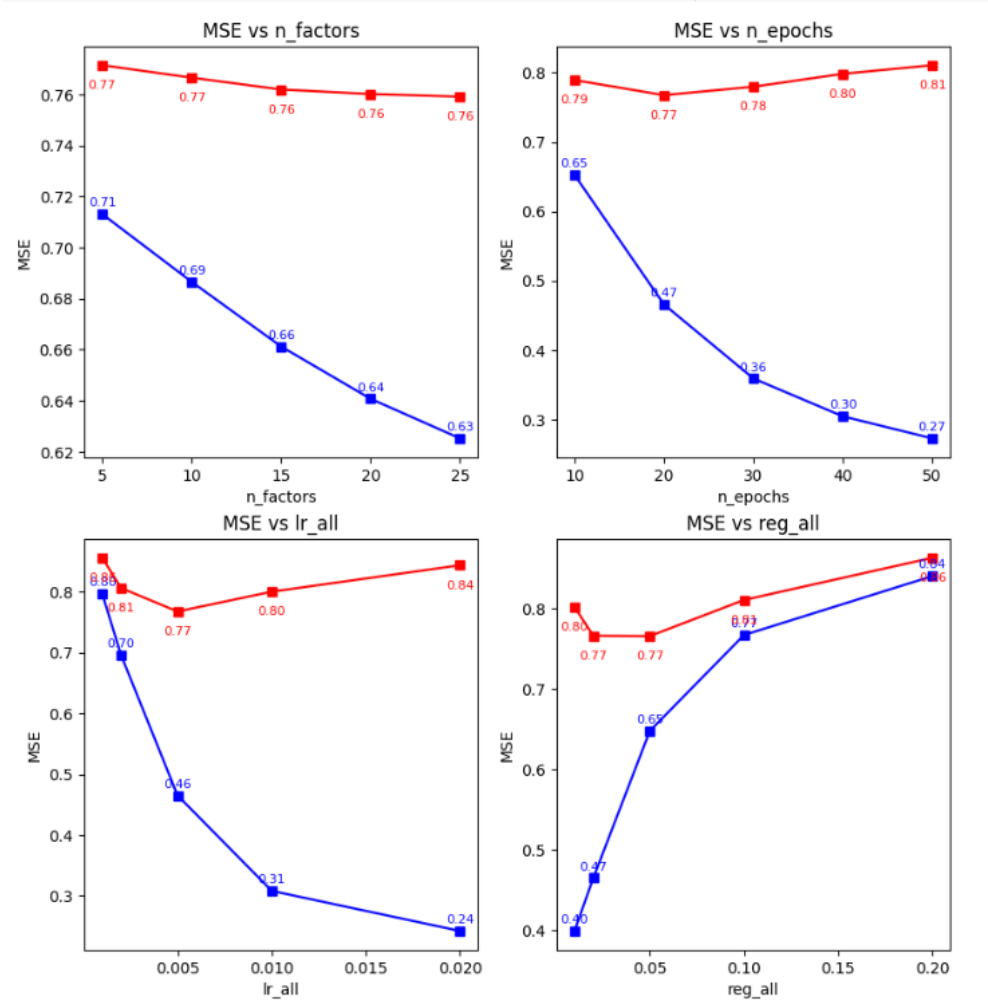
5 Experiments & Results

5.1 Singular Value Decomposition (SVD)

We used SVD from the surprise library, because it provides tools for hyperparameter tuning. Specifically, we tuned four hyperparameters, one at a time, in order to observe the effect of each hyperparameter in the Mean Squared Error. The hyperparameters we tuned are:

- **n_factors**: The number of latent factors. Increasing the number of latent factors allows the model to capture more complex patters in the data, at the expense of computational cost.
- **n_epochs:** The number of iterations the SVD algorithm will perform on the training dataset.
- **lr_all:** The learning rate of all parameters in the model. It determines how much the parameters of the model are updated in each iteration.
- **reg_all:** The regularization term for all parameters in the model. Regularization is a technique used to prevent overfitting by penalizing large values of the model's parameters.

We used five values for each one of the four hyperparameters. The execution of SVD took approximately 24,5 minutes in our local host and 16,5 minutes in our Google Colab. The results were the following:



We observe that for all hyperparameters the best value of Mean Squared Error was 0,76. The values for each hyperparameter that achieved that MSE were:

- n_factors: 15, 20, 25
- n_epochs: 20
- lr_all: 0.005
- reg_all: 0.05

At this pointed we decided to test the combinations of those values, and the best MSE at 0.7558 came as a result of the following combination of values:

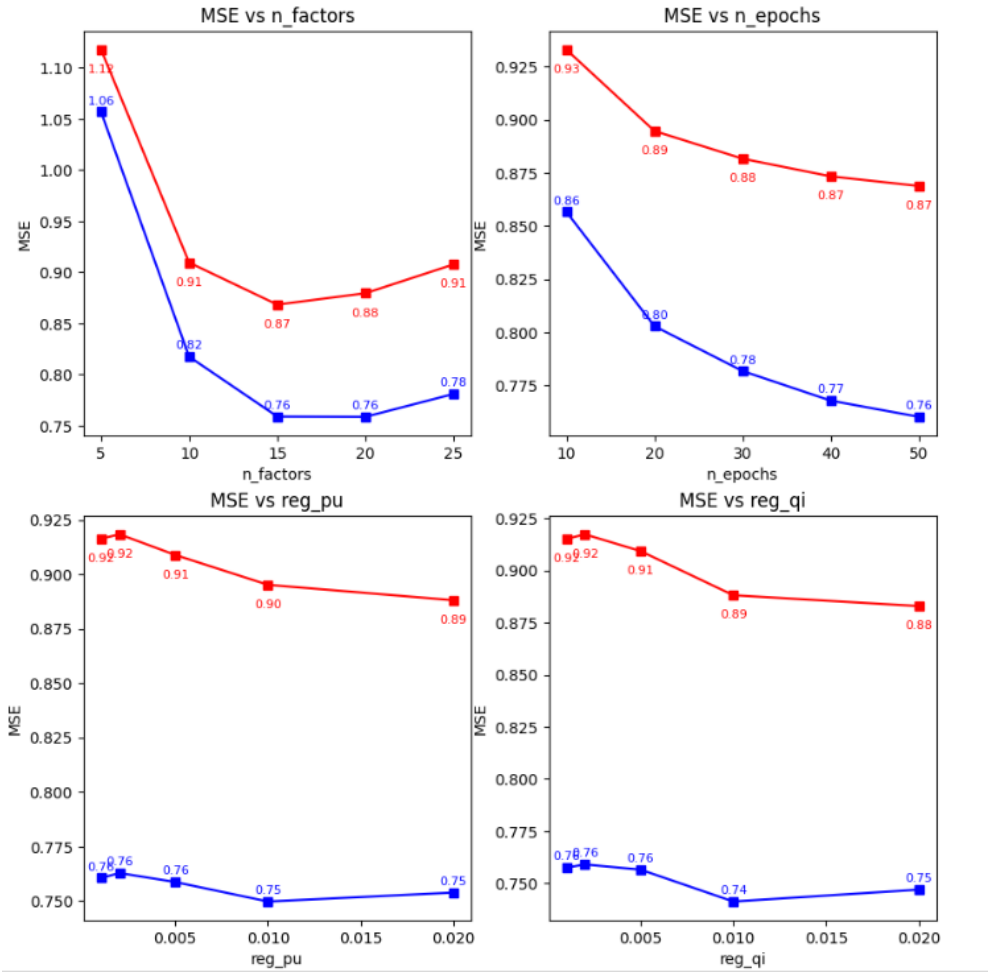
```
SVD(n_factors=25,n_epochs=20,lr_all=0.005,reg_all=0.05)
```

5.2 Non-Negative Matrix Factorization (NMF)

We used NMF from the surprise library, because it provides tools for hyperparameter tuning. We tuned four hyperparameters, one at a time, in order to observe the effect of each hyperparameter in the Mean Squared Error. The hyperparamater we tuned are:

- **n_factors:** The number of latent factors. Increasing the number of latent factors allows the model to capture more complex patterns in the data, at the expense of computational cost.
- **n_epochs:** The number of iterations the NMF algorithm will perform on the training dataset.
- **reg_pu:** Parameter that controls the regularization strength applied to the user factors.
- **reg_qi:** Parameter that controls the regularization strength applied to the item factors.

We used five values for each one of the four hyperparameters, and the results were the following:



We observe that for all hyperparameters the best value of Mean Squared Error was 0,87. The values for each hyperparameter that achieved that MSE were:

- n_factors: 15
- n_epochs: 50
- lr_all: 0.02

- reg_all: 0.02

At this point we tested the only combination of values of hyperparameters,

```
NMF(n_factors=15,n_epochs=50,reg_pu=0.02,reg_qi=0.02)
```

As a result, the MSE was observed to be 0.9190, which unlike SVD, proves that the combination of the best hyperparameter values doesn't result in an overall more efficient model.

5.3 Stochastic Gradient Descent (SGD)

In the SGD algorithm we have the following four hyperparameters: the learning rate, the lambda regularization parameter, the number of latent factors and the number of iterations that the SGD error optimization will execute. We experimented with tuning these hyperparameters by running several tests with different combinations of values for these hyperparameters. The following are some indicative results that show the effect that each hyperparameter has on the accuracy of the algorithm:

- The **learning rate** determines the step size during parameter updates. Too high of a learning rate can cause the algorithm to converge slowly or even diverge, while too low of a learning rate may result in slow convergence or getting stuck in local minima.

Learning Rate	Mean Squared Error
0.001	2.874075394668591
0.01	2.7674699165899135
0.1	2.1502139073505755
0.2	2.655999321942788

  Looking at the table above, we see that a learning rate of 0.1 yields the best results, therefore we choose that.

- **Regularization** helps prevent overfitting by penalizing large values in the latent factor matrices. Too much regularization may lead to underfitting, while too little may result in overfitting.

Lambda Regulation	Mean Squared Error
0.01	2.129491380447938
0.1	2.164994362365267
1.0	2.588200920133118

  Here, a lambda_reg value of 0.01 gives us the better result, although the difference between that and the results for the value 0.1 are relatively close.

- The **number of latent factors** represents the dimensionality of the feature space for the users and the items. Too few latent factors may lead to underfitting, whereas too many may result in overfitting.

# of Latent Factors	Mean Squared Error
5	4.625558863806613
10	2.1407417015407475
20	2.679450799139848

We choose to use 10 latent factors as it seems to be the best/most balanced choice.

- The **number of iterations** determines how many times the algorithm will loop to try and minimize the error. Too few iterations may result in underfitting and inaccurate results, while too many may lead to overfitting or slow convergence, and of course an increased execution time.

# of Latent Factors	Mean Squared Error
50	2.15074861540788970
100	1.7765457910687599
200	1.7162475745958714

Finally, 200 iterations seemed like a good enough number. The rate that the performance increases after 100 iterations drops pretty rapidly, so going that mark is not very efficient. Still, at 200, we have an acceptable execution time and decent performance.

5.4 K Nearest Neighbors (KNN)

Here, in the KNN algorithm, there is really only one hyperparameter that is worth tuning, and that is the number of neighbors (k) that we are looking for for each user-item pair. Here is how the different values affected the performance:

# of neighbors	Mean Squared Error
5	1.3051213261849535
10	1.2166817723468317
20	1.1811520802690894
40	1.165747865259876
60	1.1632004150022737

Looking at the table above we can see that the benefit of increasing the number of neighbors starts to fall off rapidly at higher values. A number of around 30 gives us a good balance of accuracy and execution time.

5.5 XGBoost

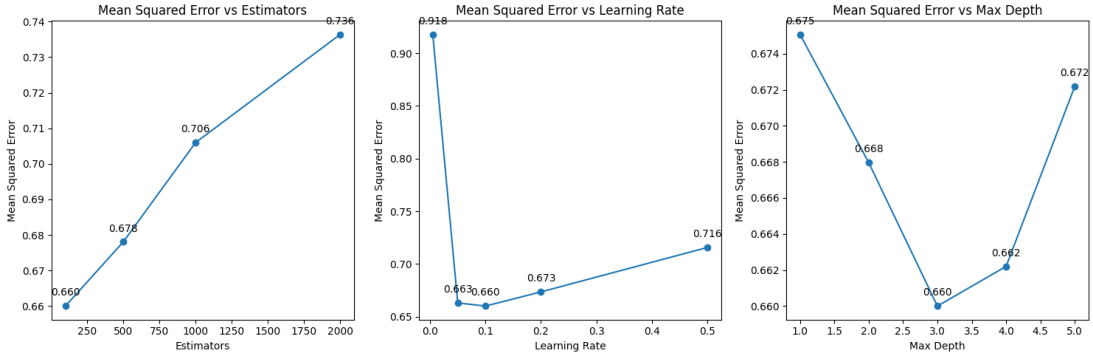
XGBoost can be used in both classification and regression problems. The recommendation system problem can be seen as a regression problem where the dependant variable is a user's rating for a certain movie.

In order for this to work, XGBoost requires that the dataset used has features. So we are going to create some features for our data. For each item we will find:

- *Average* rating of all ratings
- *Top 5* similar users who rated that movie
- *Top 5* similar movies rated by this user
- User's average rating
- *Average* rating of this movie

So, we are going to train the XGBoost model with independent variables the features we created and dependent variable the rating of a film by a user. For the evaluation of the model we are going to use *Mean Squared Error*, *Root Mean Squared Error* and *Mean Absolute Percentage Error* but we are focusing only on **Mean Squared Error** or **MSE**.

The hyperparameters we can tune are *Estimators* (number of boosting trees to build in the model), *Learning Rate* (step size during parameter updates) and *Max Depth* of the boosting trees. Here are the graphs of the hyperparameter tuning:

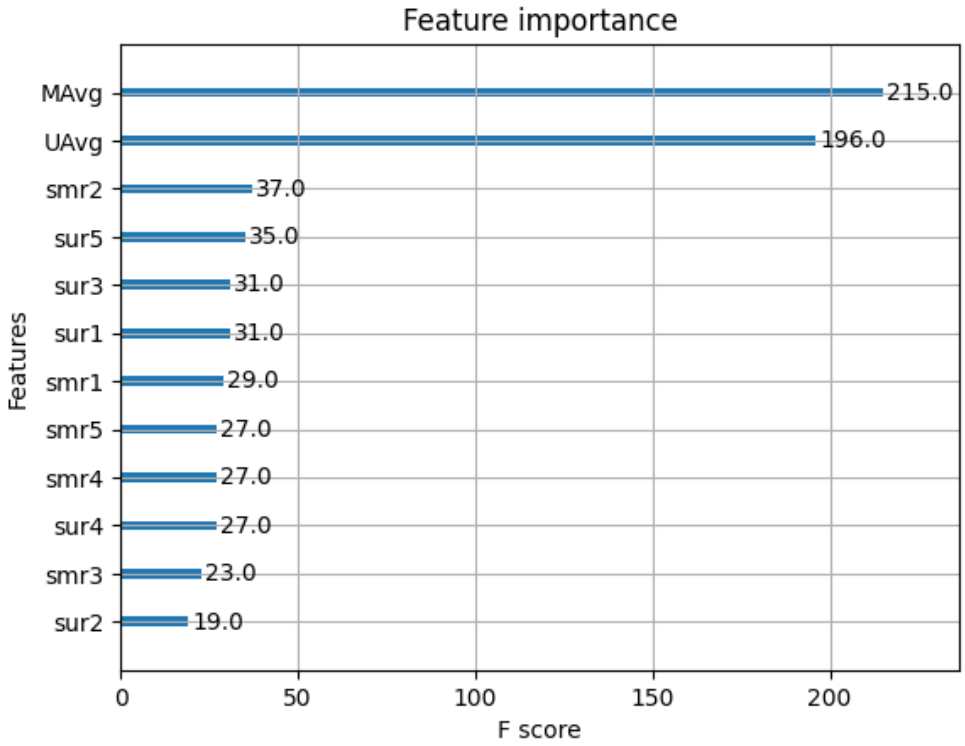


We observe that the best combination of hyperparameters to tune the model to are:

- Estimators: 100
- Learning Rate: 0.1
- Max Depth: 3

After running the model with these hyperparameters, we get a Mean Squared Error of **0.615071**. In this case, the best combination of hyperparameters gives us the lowest MSE we can get indeed. Also, this is the *lowest MSE* we have achieve in this project *so far*.

Moreover, the XGBoost algorithm also performs feature importance. This is a graph which shows the importance of each feature:



We can see that the most important features are the *average rating of the movie* and the *average rating of the user*. What we can conclude of this is that users are more likely to give a good rating to a movie that is highly rated, but also don't diverge much from their mean rating.

5.6 XGBoost + Baseline

For the Baseline model we can only do little fine tuning. The only hyperparameter that we can tune is the learning rate of the SGD model. So, let's run the baseline model for different learning rates:

Learning Rate	**Mean Squared Error**
0.001	1.184479
0.01	1.178852
0.1	1.180751
0.5	1.219276

From this table, we can see that we get better results when having the **Learning Rate** be **0.01**. Now, we are going to give the baseline model's predictions as features of the dataset given to the XGBoost algorithm and let's compare the results.

Model	**Mean Squared Error**
XGBoost	0.615071
XGBoost + Baseline	0.931935

In our evaluation of the two models, we initially anticipated that the XGBoost + Baseline model would perform better than the XGBoost model, because of the extra information it was given. However, upon closer examination, it became apparent that the second model exhibited performance metrics that were not as favorable as expected. XGBoost + Baseline's MSE is

worse than the XBoost's.

5.7 Neural Collaborative Filtering (NCF)

We are going to use *keras* in order to build a model with these layers:

User Layers

- *User Input Layer*: a one-dimensional input layer that takes in the User_ID
- *User Embedding Layer*: an embedding layer that converts the User_ID into a dense vector representation of the *Latent Features*
- *User Flattening Layer*: a flattening layer that flattens the output of the user embedding layer into a one-dimensional vector

Movie Layers

- *Movie Input Layer*: a one-dimensional input layer that takes in the Movie_ID
- *Movie Embedding Layer*: an embedding layer that converts the Movie_ID into a dense vector representation of the *Latent Features*
- *Movie Flattening Layer*: a flattening layer that flattens the output of the user embedding layer into a one-dimensional vector

Dot-Product Layers

- *Dot-Product Layer (Concat)*: takes the flattened movie and user vectors and computes their dot product. This produces a single value that represents the similarity between the movie and user

Dropout Layers

- *Concat Dropout Layer*: introduces some randomness into the model by randomly setting some of the values in the output of the concat layer to zero. This helps to prevent overfitting
- *FC Dropout Layers (3 layers)*: these layers are similar to concat dropout layer, but they are applied to the outputs of the Dense layers

Dense Layers

- *FC 1*: applies a fully connected layer with 100 units and a ReLU activation function to the output of the concat dropout layer
- *FC 2*: another fully connected layer with 50 units and a ReLU activation function
- *FC 3*: the final fully connected layer with 1 unit and a sigmoid activation function. This layer outputs the predicted rating for the movie and user

Also, the loss function we've used while training the model is *Mean Squared Error* and the optimizer is *Adam* with a learning rate of 0.01.

The hyperparameters we can tune for this neural network are the following:

- **Dimensionality** of the latent features of the embedding space of both users and movies. These features capture underlying patterns and relationships in the data, providing a more compact and informative representation. The more latent features we have, the more parameters can be trained, hence better performance.
- **Epochs** define the number of times the entire training dataset is passed forward and backward through the neural network during training. Underfitting may occur if the number of epochs is too low and, on the other hand, overfitting may occur if the number of epochs is too high.
- **Batch Size** defines the number of samples used in each iteration during training of a neural network. In other words, it determines how many data points are processed before the model's weights are updated. Batch size is a critical parameter that influences the training dynamics, convergence speed, and memory requirements of a neural network.

Speaking of **dimensionality** of the latent features, we will try running the model for different latent dimensions and see which gives the better result.

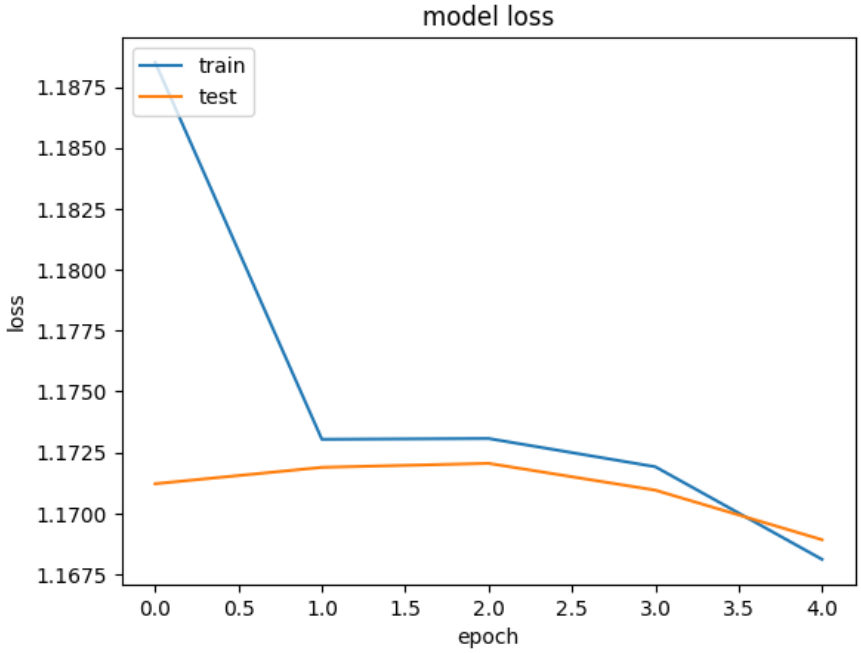
Dimensions	Trainable Parameters	Training Time	Mean Squared Error
10	278,321	0:08:42	1.173863
20	551,341	0:09:22	1.173746
50	1,370,401	0:08:26	1.171590
100	2,735,501	0:09:06	1.174202

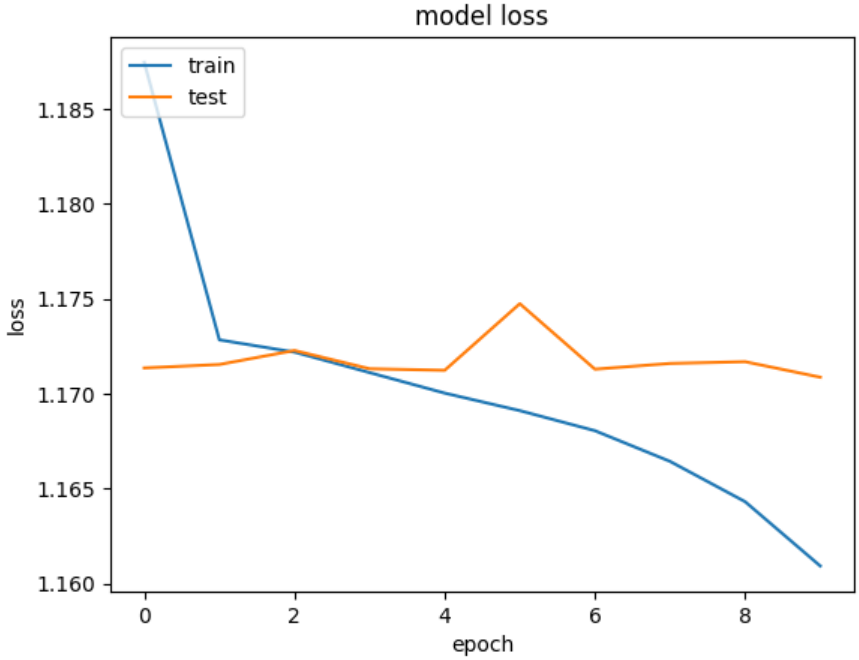
Looking at the table above, we can conclude to the following: the training time doesn't change much when increasing the dimensions, the trainable parameters are proportional to the dimensions. Also and most important, the MSE stays around the same, something that we didn't expect. However, when the latent dimension are **50**, we get a slightly better MSE (**1.171590**).

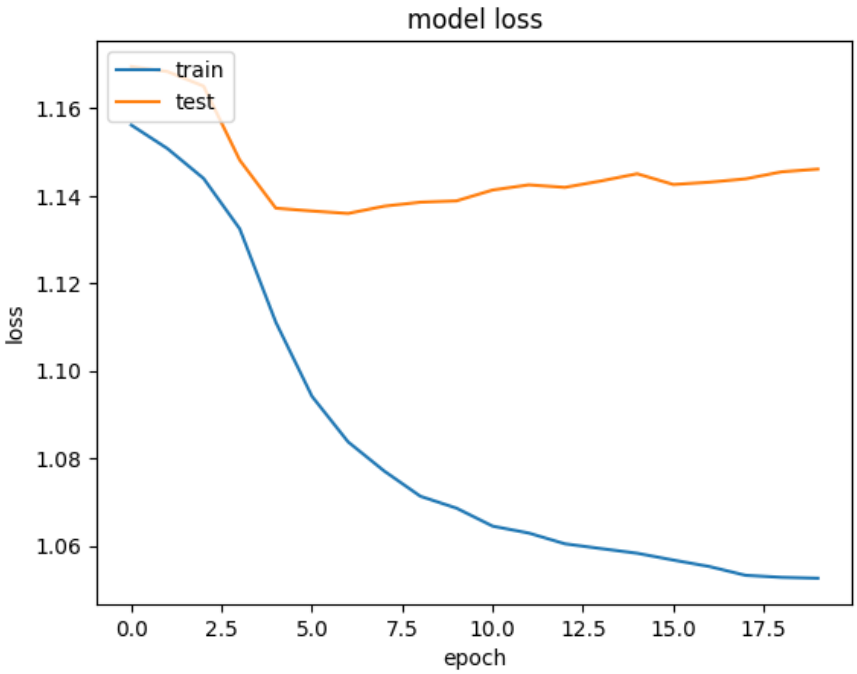
So now, we'll train the model with 50 latent dimension but for even more **epochs** to see if we'll get a better MSE. The results we get are these:

Epochs	Training Time	Mean Squared Error
5	0:08:26	1.171590
10	0:17:19	1.173537
20	0:35:07	1.148805

As it was expected, with the increase of epochs, the model needs more time to train. However, while we expected the MSE to drop, it remains the same. Let's examine the *model loss* graphs for different epochs. These plots show the loss of both train set and test set of each epoch.


model loss
train
test
1.1875
1.1850
1.1825
1.1800
1.1775
1.1750
1.1725
1.1700
1.1675
loss
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
epoch


model loss
train
test
1.185
1.180
1.175
1.170
1.165
1.160
loss
0
2
4
6
8
epoch



We can see that while training the model, at first both train set loss and test set loss is dropping. However, after a certain point, the train set loss continues dropping, but the test set stays the same around a certain value and then starts increasing. That means that the model is *overfitting* after 5 or so epochs. So, one can say that the optimal number of epochs is 5.

We observe that the best combination of hyperparameters to tune the neural network to are:

- Latent Dimensions: 50
- Epochs: 5
- Batch Size: 64

After running the model with these hyperparameters, we get a Mean Squared Error of **1.166544**.

6 Conclusions & Future Work

In conclusion, we tested a variety of models, from matrix factorization techniques to neural networks. We evaluated each model's performance using the Mean Squared Error (MSE). Specifically, the MSE of every model was,

- **Singular Value Decomposition (SVD): 0.756**
- **Non-Negative Matrix Factorization (NMF): 0.870**
- **Stochastic Gradient Descent (SGD): 1.716**
- **K Nearest Neighbors (KNN): 1.163**

- **XGBoost: 0.615**
- **XGBoost + Baseline: 0.931**
- **Neural Collaborative Filtering (NCF): 1.167**

After a thorough examination of our models, we conclude that **XGBoost** outperformed the other models, achieving the minimum Mean Squared Error of **0.615**. It's also worth noting that all the models performed well and in a similar manner, although some models, like XGBoost + Baseline, didn't perform as well as we might have expected.

There are several methods that can be applied to improve our movie recommendation system. Firstly, the cold-start problem can be addressed and a solution to it can be proposed, leveraging user and movie characteristics and performing content-based filtering in order to find similar users and similar movies, something that our dataset didn't support. Moreover, when experimenting with baselines on the XGBoost model, there are several more baseline models we can be used, like KNNBaseline, and overall we all the XGBoost based models can be combined so we could have ultimate results. Finally, metrics like the timestamps of the ratings can be taken account in order to make time-sensitive recommendations and explore how user preferences change over time and adjust the recommendations accordingly.

7 Contributions

Each team member worked on:

- Apostolos Falaras, 2981: Exploratory Data Analysis (EDA), Singular Value Decomposition (SVD), Non-Negative Matrix Factorization (NMF).
- Maipas Kosmas, 2567: Stochastic Gradient Descent (SGD), K Nearest Neighbors (KNN), Python Recommender Application Implementation.
- Sialtsis Alexandros, 3083: Featurizing Data, XGBoost, XGBoost + Baseline, Neural Collaborative Filtering (NCF)

References

[1] S. Agrawal and P. Jain, "An improved approach for movie recommendation system," 2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), Palladam, India, 2017, pp. 336-342, doi: 10.1109/I-SMAC.2017.8058367.

[2] J. Zhang, Y. Wang, Z. Yuan and Q. Jin, "Personalized real-time movie recommendation system: Practical prototype and evaluation," in Tsinghua Science and Technology, vol. 25, no. 2, pp. 180-191, April 2020, doi: 10.26599/TST.2018.9010118.

[3] He, X., Liao, L., Zhang, H., Nie, L., Hu, X., & Chua, T. S. (2017, April). "Neural collaborative filtering". In Proceedings of the 26th international conference on world wide web (pp. 173-182).

[4] Y. Koren, R. Bell and C. Volinsky, "Matrix Factorization Techniques for Recommender Systems," in Computer, vol. 42, no. 8, pp. 30-37, Aug. 2009, doi: 10.1109/MC.2009.263.

[5] Chen, R., Hua, Q., Chang, Y., Wang, B., Zhang, L., & Kong, X. (2018). "A Survey of Collaborative Filtering-Based Recommender Systems: From Traditional Methods to Hybrid Methods Based on Social Networks". IEEE Access, 6, 64301-64320.

[6] Shahbazi, Zeinab, and Yung-Cheol Byun. "Product recommendation based on content-based filtering using XGBoost classifier." Int. J. Adv. Sci. Technol 29 (2019): 6979-6988.

[7] Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.

[8] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[9] François Chollet and others. (2015). "Keras." Available: https://keras.io

[10] Hug, N., (2020). Surprise: A Python library for recommender systems. Journal of Open Source Software, 5(52), 2174, https://doi.org/10.21105/joss.02174