What to watch next? A comparative analysis of AI approaches for movie recommender agents

36495, 34611, & 31241 (Candidate IDs)

Abstract

In this project we explore how one can design an AI agent that can recommend movies to the user by using learning approaches. Using the MovieLens dataset we compare several AI approaches. Starting with the most simple cases, e.g. predicting how to recommend movies based on genre similarities, our final agent function takes into account both latent features of the movies and the implicit choices of the users. We conclude, that there is still many challenges ahead to design a good movie recommendation agent, but that it is important to keep in mind that the human agents (e.g. experts) often fail in that respect too.

1. INTRODUCTION

1.1 MOTIVATION: WHAT TO WATCH NEXT?

Imagine that it is Friday and you want to watch a good movie. You choose a movie based on a review in a newspaper, but unfortunately the movie is not to your taste. Instead you choose one you know. You feel bored to watch the same movie again, and keep wondering why the human agent did not succeed, and if an artificial intelligent agent could do better?

1.2 PROBLEM STATEMENT: A COMPARATIVE APPROACH

In what ways can one design an artificial intelligent agent that can help the user pick a good movie perceiving the environment and learning from the users' taste?

1.3 THE AI AGENT AND ITS TASK ENVIRONMENT

Let us start with a definition:*an agent is anything that can be viewed as perceiving its environment through sensors and acting upon the environment through actuators* (Russell and Norvig, 2020). In other words, an AI movie recommender **agent** faces an **environment** of all possible movies, in which the agent **perceives** what the user rates, and **learns**, by sensing the users' behaviour, how to **act**, i.e. what to recommend.

The first important task is to define the **task environment** for our **software agent** using the **PEAS framework**. See below.

Agent type	Performance measure	Environment	Actuators	Sensors
Movie recommender	User rates movie	All movies available	Suggest movies	Movie features, user history

Table 1: PEAS description of the task environment

In broader terms, the environment is characterized by being **partially observable** (and thus, **unknown**), because no matter the agent function, only *some of* the relevant features of the movies and the users' taste are available. There is only a **single agent** in the environment (the other users are not agents, as their actions will not affect the performance measure), but the agent can of course learn from and recommend movies to multiple users. The environment is, in most cases, **sequential**, **non-deterministic**, **semi-dynamic**, and having a **continuous statespace**. In layman terms: A movie recommendation agent is not going to have an easy job (but still easier than e.g. driving a car - and less risky to fail!).

The task environment for a movie recommendation agent is not universal, instead is dependent on the available data, why the PEAS description might need to be altered if one for instance was designing an AI agent for Netflix instead of using the MovieLens Dataset. In a case like that, the performance metric would be improved because we could get data on the movies that the user had clicked on, watched partly or completely, etc.
In our case, we have data about the movies (e.g. genres) and **percept sequences** of the users' ratings of the movies. The more the users interact with the environment, the more our agent can learn. For that reason, the biggest challenge for a movie recommender agent is new movies and new users, where there is yet no **percept sequence** available.

2. DATA

2.1 THE RELEVANT DATA AND AGENT FUNCTIONS

There are two types of data relevant to train a movie recommender agent: Data about the **items** (the movies) and about the **users**. The data about the movies could be features like titles and genres, whereas the data about the users is about their preference for movies. Here, one can distinguish between explicit and implicit data.

In this project, we mainly use explicit data about the user's preference - their ranking of a particular movie. This is for obvious reasons more difficult to obtain than data about the movies themselves (it exists in public databases like IMDB), but it is available in the MovieLens Database. Other examples of explicit data, that was not available, are whether a user watched a particular movie or was surveyed about her opinion of a particular movie. In reverse, implicit data is not given explicitly by the user. Using implicit data the agent could learn from proxies, e.g. from the ratings that the user didn't give, or by data collected from secondary sources (clicks, purchases). We will explore one approach to implicit data.

2.2 THE MOVIELENS DATASET

We use the *MovieLens 100K Dataset* that consist of 100,000 ratings from 1000 users on 1700 movies (Harper and Konstan, 2015). This data was first released in April, 1998. For the purpose of this comparative project, it has a sufficient and computaionally efficent size. The dataset consist of 2 tables with movies (i.e. items) and ratings (i.e. users) respectively. The movies table contains the movie id, title, and genre while the ratings table contains the timestamp, user id, movie id, and ratings. The average ratings given by users to movies is 3.5 while the minimum and maximum rating given is 0.5 and 5, respectively.

3. TYPES OF AGENT FUNCTIONS FOR RECOMMENDER AGENTS

In the following we will look at different kinds of agent functions for a movie recommendation system, and the report is structured like the table below. We will look at two types of models for the agent functions: Neighborhood Models and Latent Factor Models. **Neighborhood models** give recommendations based on similarity measures. It works by finding similar movies (the most simple approach) or by similar users. The more advanced **Latent Factor models** work by finding hidden features in the movies that the users are ranking, and are thus (just) more advanced collaborative filtering approaches (Hu et al., 2008).
We will implement four of these functions and our approach is to start with the most simple agent function and gradually consider more complex agent functions.

	Data	**Feedback data**	**Model type**	**Agent function**
4.1	Movie	Explicit	Neighbourhood model	"Content-based filtering"
4.2	User	Explicit	Neighbourhood model	"Simple Collaborative filtering"
5.1	User-Movie	Explicit	Latent factor model	"Singular Value Decomposition"
5.2	User-Movie	Implicit/Explicit	Latent factor model	"Alternating Least Squares"
5.3	User-Movie	Implicit/Explicit	Latent factor model	"Bayesian Personalized Ranking"

Table 2: Comparison of agent function characteristics

4. NEIGHBOURHOOD MODELS

4.1 CONTENT BASED FILTERING VIA SIMILARITY

An example of a content based recommender system is an agent function that makes decisions based on the genres of the movies. It perceives that the user like a particular movie of a genre, say "Thriller", and recommend another movie of that genre. This concepts works even better the agent takes combinations of genres.

4.1.1 THE AGENT PROGRAM - IMPLEMENTATION OF THE MODEL

The agent program uses a vector representation for each movie so that it can compare if 2 movies are similar or not. Importance is given to having a rare genre, i.e, if a person likes a genre that is not very common, finding another movie with that genre would make a much better match than to find movies with other genre matches.

For a movie having combination of common genres (e.g "Action") and uncommon genres (e.g "War"), more importance can be given to the uncommon genres. A common approach for this problem is to use the **Term Frequency-Inverse Document Frequency** (TF-IDF) Vectorizer. *TF-IDF is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus* (Wikipedia, 2022). It helps capture the important genres of each movie by giving a higher weight to the less frequent genres.

$$w_{i,j} = tf_{i,j}.log\frac{N}{df_i}$$

where, $w_{i,j}$ is the weight for an item, $tf_{i,j}$ = number of occurrences of i in j, $df_{i,j}$ = number of documents, and N = Total Number of documents

The function take combinations of up to four genres, i.e if we have "Adventure,Comedy,Action", we are taking combinations like "Adventure","Comedy", "Action", "Adventure, Comedy", "Comedy, Action" and so on, of course such that "Comedy, Action" and "Action, Comedy" are treated the same, *since the order of the genres is not relevant, we want to account for the combinations of genres for a given movie, regardless of the order.* (Nixon, 2020) Then we calculate the TF-IDF weights for each of the movies using these combinations. Finally, we calculate the similarity between the movies by using the Cosine Similarity.

If the genres are a perfect match, for example, the genre(s) for Toy Story (1995) obviously perfectly match with itself, the similarity score is 1, and if the genres don't match at all, (Toy Story (1995) obviously doesn't match with Sudden Death (1995)), then the similarity score is 0. Therefore, given any movie, top recommendations have the highest similarity scores with that particular movie.

4.2 SIMPLE COLLABORATIVE FILTERING VIA SIMILARITY

The motivation behind this collaborative filtering is that if user 1 likes movies A, B and C and if user 2 likes movies A, C and D, then, user 1 might like movie D and user 2 might like movie B. This agent function recommends movies based on similarities between users and what other users have liked. We have considered 2 different agent functions.

In the first and item-based approach, the recommendations are given based on how close one movie is to another based on the kind of ratings they have gotten from the users. What we want is to find a way to measure the closeness between different movies and then based on one movie, the agent recommends the closest k movies. Basically, if the movies were treated like points on a graph, we assume that the points closest to a movie are similar to that movie, and hence are something that the user might like.

In the second and user-based approach, the agent function calculates the rating that a user u would give movie m, instead of directly making recommendations. This technique recommends movies to a particular user based on other users whom they are most similar to. This similarity among users is calculated based on ratings (not other factors like age, gender etc. - that the agent cannot perceive). Once the set of similar users are identified, then ratings are predicted for previously unseen movies and the top rated movies are recommended. The similarity between users can be calculated in 3 ways, using Euclidean Distance, Cosine Similarity or Pearson Correlation Coefficient.

4.2.1 THE AGENT PROGRAM - IMPLEMENTATION OF MODEL

To fit the K-Nearest Neighbor algorithm, we need a mxn matrix where m is the number of movies and n is the number of users. We can use the pivot_table() command in Python to achieve this. We also fill any missing values with 0 as this will help us when we calculate the distances between points. The matrix thus formed will be a very sparse matrix and

therefore, to make the calculation more efficient in Python, we convert it into a scipy sparse matrix (Liao, 2018).

If we were to fit the resulting matrix directly to a KNN Algorithm, it could suffer from **Curse of Dimensionality**. This is because our matrix has too many features. KNN, by default, uses **Euclidean Distance** to calculate the distance between 2 points, and with so many features in our data, we can assume that for any given point, the Euclidean Distance of the majority of other points will be approximately same. This is not helpful when we want to make recommendations. What we do instead is, use Cosine Similarity to calculate the *distance*. While fitting the KNN Algorithm, the value of k is the number of recommendations we want for each movie.

To implement the collaborative filtering recommender system where ratings are predicted, in python, the *Surprise* package (Hug, 2020) is used. The package gives us a direct implementation of K-Nearest Neighbors algorithm for calculating ratings. 3 different variations of the KNN algorithm is considered - **KNNBasic**, **KNNMeans** and **KNNZScore**. The 3 different models are trained and tested on the test data. The one with the lowest rMSE score was chosen and *Grid Search* is done to find the optimal parameters for the similarity measure and the minimum support value.

$$
\begin{aligned}
&\text{KNN Basic:} && r_{u,m} = \frac{\Sigma_{v \in N(u)} sim(u,v) r_{v,m}}{\Sigma_{v \in N(u)} sim(u,v)} \\
&\text{KNN Means:} && r_{u,m} = \mu_u + \frac{\Sigma_{v \in N(u)} sim(u,v)(r_{v,m} - \mu_v)}{\Sigma_{v \in N(u)} sim(u,v)} \\
&\text{KNN Z-Score:} && r_{u,m} = \mu_u + \sigma_u \{ \frac{\Sigma_{v \in N(u)} sim(u,v) \frac{(r_{v,m} - \mu_v)}{\sigma_v}}{\Sigma_{v \in N(u)} sim(u,v)} \}
\end{aligned}
$$

Here, $N(u)$ is the set of similar users to u found using KNN, $sim(u, v)$ is the similarity score between user u and v, $r_{v,m}$ is the rating given by user v to movie m, μ_u is the mean rating by user u and σ_u is the variance of the ratings by user u.

5. LATENT FACTOR MODELS (COLLABORATIVE FILTERING)

Wouldn't it be nice if the movie recommender agent could take into account latent features as well? Users may also have their preferences for each of these features like movies with action or with a dog as part of the plot, that can not be observed directly. Say, User A tends of give high ratings to movies that have a dog in them, while User B prefers movies with actions. If a Movie X is a action movie with a dog, both User A and B would give high ratings. Since not all users have watched all movies, we would like to find a way to identify the various features a particular movie has, and also how much does a certain user prefer these features based on the ratings given by the user to movies with these features. With this information, the agent would be able to predict what rating a particular user will give to a movie they haven't yet watched, depending on their preference of the features of the movie. It can then, recommend a user movies that it predict would get high ratings from the user.

5.1 MATRIX FACTORIZATION (SINGULAR VALUE DECOMPOSITION)

Matrix Decomposition helps us do this. Matrix Factorization is the process of decomposing a user-item interaction matrix into a product of lower dimensionality matrices. It allows a large matrix to be broken down to its constituent parts making the calculation of more complex matrix operations much simpler and efficient. If we take the matrix showing the association between the users and the movies, say R, through matrix decomposition, we can find constituent matrices having lower dimensionality, whose product is R. Since not all users have seen all movies, multiple values in R would be missing. We try to find the *best guess* for the constituent matrices so that the error between the non-missing values in R and the corresponding values in the product of the constituent matrices is minimized. Once we have the constituent matrices that minimize this error, their matrix product would contain the predictions for the missing values in R.

5.1.1 THE AGENT PROGRAM

We work with the user-item matrix, R, containing the users as rows and movies as columns. The values in the matrix are the ratings given by the user to the corresponding movie. We fill any missing values with 0. We also, further, normalize the data.

We then perform Matrix Decomposition through the **Singular Value Decomposition (SVD)** method. SVD is an algorithm that decomposes the matrix R into the best lower rank approximation of the original matrix R. This is done by breaking down R in the following way: $R = U\Sigma V^T$ where U and V are unitary matrices and U and V^T are orthogonal. Σ is a diagonal matrix containing singular values, which depict the weights. We can interpret U as the matrix depicting the association between the *Users* and the *Features*, or in other words, how much does each User *like* each Feature. V^T, on the other hand, can be interpreted as how relevant each Feature is to each Movie (Becker, 2016).

To perform SVD in Python, we make use of the *svds()* function from Scipy. We have to specify the value for the number of latent features we wish to consider, k. This value can be optimised through train-test-validation methods of minimizing the Root Mean Square Error (rMSE). Increasing k decreases the error on the training set, but can easily lead to overfitting, and in turn increase the error on the testing set. It is found that values of k between 20 and 100 work the best.

Once we perform SVD, for any user, we can see what movies they have already rated, and use our predictions to find the movies which the user will most likely give a high rating to from the set of movie that he/she has not already seen. These form good recommendations for that particular user.

5.2 MATRIX FACTORIZATION (ALTERNATING LEAST SQUARES)

In this approach, each movie is characterized by a preference value and its confidence. Initially, missing values is considered to have negative preference with a low confidence value and rated movies to have a positive preference with high confidence. The confidence can be calculated by using factors like the number of times the user has watched that movie or

any other form of interaction.

The preference is a binary value, 1 for positive and 0 for negative. The confidence is calculated as a linear function of the ratings given by the user: $c_{u,i} = 1 + \alpha r_{u,i}$
Here $r_{u,i}$ is the rating given by user u to movie i and $c_{u,i}$ is the corresponding confidence value. Here, α is a linear scaling parameter.

The aim of this matrix factorization is to decompose the ratings matrix R into 2 matrices U and V such that $R = UV$. Here, U and V are the user and movie matrices with hidden features (latent features). Least squares method is used to find the best approximation of R. It is called alternating least squares because in each iteration, one of U or V is kept constant and the other is optimized using regularization (Hu et al., 2008). This can be implemented using the implicit package, but we will focus on the next agent function:

5.3 MATRIX FACTORIZATION (BAYESIAN PERSONALIZED RANKING)

This last agent function aims to come up with even more personalized rankings for users rather than predicting whether the user will watch a particular movie or not. For this, the training data is considered in terms of pairs of items for each user, (u, i, j), where u is the user, i is considered to be a 'positive' movie and j to be a 'negative' movie. Here, 'positive' and 'negative' is defined in the sense whether the user has interacted with that particular movie. So if u has rated movie 1 but not movie 2, we say that movie 1 is a 'positive' movie and movie 2 as a 'negative' movie for user u. But we can't say anything between pairs of movies that have both been rated or not rated (Rendle et al., 2012).

The method optimizes the following equation: $\Sigma_{(u,i,j)\in D_s} ln(\sigma(\hat{x}_{uij})) - \lambda_\Theta ||\Theta||^2$
Here, Θ is the matrix factorization model parameter like the user and movie matrices. D_s is our data set containing all the interactions between users and movies in the form of (u, i, j). $\hat{x}_{uij}$ is a function that represents the relation between user u, movie i and movie j . This is calculated using matrix factorization and then passed into the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$ that returns the probability of user u preferring movie i to movie j . This is optimized using the regularization parameter λ_Θ. The matrix factorization method in this model aims to decompose $\hat{x}_{uij}$ as $\hat{x}_{ui} - \hat{x}_{uj}$.

Optimising the Bayesian Personalized Ranking criterion is the similar to optimizing the AUC (Area Under the Curve) metric, that is a rank based criterion. The optimization process is implemented through gradient descent using bootstrapping where in the movie j is chosen randomly for faster convergence.

This method was implemented using the *LightFM* package (Kula, 2015) in python.The MovieLens data set is in the package and split into train and test data. The model is fit on the training data with number of latent features as 15 with and a learning rate of 0.05.

6. MOVIE RECOMMENDATIONS AND AGENT COMPARISON

We have implemented four different agent functions as agent programs, starting from the simplest approaches and ending with the most complex, that are able to sensor and learn more and more from the environment. Here we go through the agents' recommendations.

The first two agent functions are neighbourhood models and similarity-based. They are both simple and intuitive agents acting on either the movies or the user ratings. The easiest to implement is content based filtering since it is item-specific and gives recommendations based on similar genre movies. Since the agent perceives nothing about the user's taste in terms of ratings, the next agents are able to use more information from the environment. The simple collaborative filtering (whether item- or user-based) takes into account the users' ratings, which gives other and potentially better recommendations, especially in terms of recommending movies outside the specific genres.

In the table below is the top 5 recommendations for the movie *Aladdin (1992)* using these two techniques. It is clear that the agent to the left is indeed recommending movies of similar genres, where the agent to the right is broader in this sense. Using common sense both agents' recommendations make sense and might even supplement each other, for instance in the cold start case, where either the movie or the user is completely new with no percept sequence available. The performance metric (more ratings) will help the agents learn what is the taste of the user.

Content based filtering (4.1)	**Item-based Collaborative filtering (4.2)**
Oliver & Company (1988)	Beauty and the Beast (1991)
Hercules (1997)	The Lion King (1994)
Robin Hood (1973)	Jurassic Park (1993)
Pete's Dragon (1977)	True Lies (1994)
Song of the South (1946)	Batman (1989)

Table 3: Examples of recommendations from the first two similarity based agent functions

To improve our movie recommendation agent even further, we implement agent functions in the category of latent factors models, which takes into account both information about the users and the movies. These also apply the logic of collaborative filtering and are all based on the method of matrix factorization. These agent functions observe even more of the environment, that is why the agents are even better in helping the user discover new movie interests. Still, the singular value decomposition recommendation (SVD) agent, are challenged by the cold start problem (described above) and the fundamental problem of self-selection bias (since ratings are voluntary, and the only explicit metric that we have to reveal what the user likes).

To overcome these issues we are interested in taking into consideration implicit feedback data about the users' preferences as well. We introduce the idea of the Alternating Least Squares agent function, that could help overcome these issues, but the problem is that this would not be personalized for each user - and that is fundamental for our agent! Our final

agent and most advanced agent is for that reason using Bayesian Personalized Rankings, that predicts the user's preference for all movie pairs. In this way, the agent learns not only from the explicit rankings, but also from the movies that were not ranked. This is a very important improvement, because the decision to rank something contains more information than the numerical ranking itself: It tells the agent that this movie is of more interest than the large amount of movies the user never interacted with.

In the table below we compare the recommendations from the two matrix factorization models with the simple user-based collaborative filtering approach for a random user (#10)[1]. The recommendations are not similar, but all reasonable and they show once again how the agent functions could potentially compliment each other. There might not be one good movie recommendation agent, but several with different strengths and weaknesses.

User-based Collaborative filtering (4.2)	**SVD (5.1)**	**BPR (5.3)**
Clockers (1995)	Pirates of the Caribbean	Raiders of the Lost Ark (1981)
Anne of Green Gables (1985)	Inception (2010)	The Empire Strikes Back (1980)
When Worlds Collide (1951)	Harry Potter III (2004)	Star Wars (1977)
The Wedding Ringer (2015)	Harry Potter I	Indiana Jones (1989)
The Book of Life (2014)	Monsters, Inc. (2001)	Groundhog Day (1993)

Table 4: Examples of collaborative filtering recommendations by AI agents

7. CONCLUSION

This project aimed to explore and compare ways of designing movie recommendation agents; from the simplest neighbourhood agent functions to the most complex latent factor learning agent functions. We conclude that there is no agent to rule them all, instead they all have their strengths and weaknesses. To radically improve the agents, one would need agents able to perceive more of the environment instead of just the movies the users ranked. This could points towards other data sources than the MovieLens dataset, data-centric AI-approaches or a combination of the movie recommender agents (with the own distinct agent function) as parts of a multi-agents recommender system, where they come up with different recommendations and need to agree on the best recommendations. There is still much to improve on for the AI agents, but keep in mind how the human agents are doing!

	Agent function type	**Strengths**	**Weaknesses**
4.1	Content-based filtering	Intuitive and easy to implement Works for movies with no ratings Can recommend niche items	Limited ability to expand based on user's interest No importance given to ratings
4.2	Simple collaborative filtering via similarity	User data is taken into account Importance given to ratings	New movies have no ratings Time complexity
5.1 (5.2) 5.3	Collaborative filtering via matrix factorization	User and Movie data is taken into account Latent features learned by the model Help users discover new interests Can include implicit feedback data (BPR)	New movies/users have no ratings Time complexity

Table 5: Comparison of the different approaches for a AI movie recommender agent

1. User-based collaborative filtering technique: RMSE score on the test set is 0.9004.
BPR: AUC on the training set is 0.9 and on the test set is 0.86.

REFERENCES

Nick Becker. Matrix factorization for movie recommendations in python, Nov 2016. URL https://beckernick.github.io/matrix-factorization-recommender/.

Denise Chen. Recommendation system-matrix factorization, Jul 2020. URL https://towardsdatascience.com/recommendation-system-matrix-factorization-d61978660b4b.

F. Maxwell Harper and Joseph A. Konstan. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4), dec 2015. ISSN 2160-6455. doi: 10.1145/2827872. URL https://doi.org/10.1145/2827872.

Yifan Hu, Yehuda Koren, and Chris Volinsky. Collaborative filtering for implicit feedback datasets. In *2008 Eighth IEEE International Conference on Data Mining*, pages 263–272, 2008. doi: 10.1109/ICDM.2008.22.

Nicolas Hug. Surprise: A python library for recommender systems. *Journal of Open Source Software*, 5(52):2174, 2020. doi: 10.21105/joss.02174. URL https://doi.org/10.21105/joss.02174.

Maciej Kula. Metadata embeddings for user and item cold-start recommendations. In Toine Bogers and Marijn Koolen, editors, *Proceedings of the 2nd Workshop on New Trends on Content-Based Recommender Systems co-located with 9th ACM Conference on Recommender Systems (RecSys 2015), Vienna, Austria, September 16-20, 2015.*, volume 1448 of *CEUR Workshop Proceedings*, pages 14–21. CEUR-WS.org, 2015. URL http://ceur-ws.org/Vol-1448/paper4.pdf.

Kevin Liao. Prototyping a recommender system step by step part 1: Knn item-based collaborative filtering, Nov 2018. URL https://towardsdatascience.com/prototyping-a-recommender-system-step-by-step-part-1-knn\-item-based-collaborative-filtering-637969614ea.

Àlex Escola Nixon. Building a movie content based recommender using tf-idf, Aug 2020. URL https://towardsdatascience.com/content-based-recommender-systems-28a1dbd858f5.

Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. BPR: bayesian personalized ranking from implicit feedback. *CoRR*, abs/1205.2618, 2012. URL http://arxiv.org/abs/1205.2618.

Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson, 2020. ISBN 9780134610993. URL http://aima.cs.berkeley.edu/.

Wikipedia. Tf–idf, Jan 2022. URL https://en.wikipedia.org/wiki/Tf%E2%80%93idf.