Comparing Inductive Recommendation Methods – Matrix Factorization vs Deep Graph-based Matrix Completion

0 - Introduction

The purpose of this project was to compare the performances of two distinct recommendation engines that can work inductively.

Typically, a recommendation system needs to be retrained when new users are added to the overall pool of users, taking a lot of time and making incremental results inefficient. However, methods exist that allow a trained model to make predictions for new users without having to retrain, using only the partial data gathered from these new users. This class of models are referred to as inductive recommendation systems.

The use case for inductive recommenders is as follows:

Let's say that we train a large model that provides recommendations/predictions, and want to allow people to access this model via a web or mobile app. For a non-inductive model, the options would either be to retrain the entire model for each new user, which simply wouldn't be feasible, or to require a signup from each user, where the model would keep track of their preferences and update recommendations to the entire base periodically. However, if the user wanted near-instant predictions after providing a small amount of information necessary for the model to operate effectively, the model would need to be inductive.

In this project I explored two different inductive models, a vanilla Matrix Factorization model with an update using the one-sided least squares algorithm, and a graph based matrix completion algorithm called IGMC. These two methods are compared alongside each other via two metrics; RMSE and MAP (mean average precision, a common metric used for recommenders).

As the dataset for this project, I will be using user-anime-rating triples that I have personally scraped from the myanimelist.net website. The site is sufficiently accessible that I was able to write custom code to scrape all the necessary data, in addition to having a large and varied userbase. Therefore, the anime dataset I gathered is very similar structurally to the famous Movielens/Netflix dataset for movies.

1 - Data Collection (datacollection.py)

First, I needed to write code to collect the data. I used the urllib and BeautifulSoup Python libraries to write my web-scraping code. I wrote two separate Python classes, one for collecting anime series and their metadata, one for collecting user ratings for anime.

To collect the anime, my code scraped data from the 'top anime' list of myanimelist.net, getting all the data including release date, genre, and average rating for approximately the top 5000 anime series.

For the user data collection, I first collected a group of random users from myanimelist.net, and then iterated over the profiles of all of these users, collecting their ratings of each anime series they had watched from the webpage HTML. I needed to interact with the webpages to scroll down the entire user lists, so I used a Chrome web driver to accomplish this.

Some users don't provide scores to the shows they've watched, so I implemented a minimum number of rated shows in order to exclude users who wouldn't be helpful for training the model.

In total, the collected data contained over a million user-anime-rating triples, with over 5000 anime series and over 8000 individual users.

2 – Data Cleaning and Preprocessing (preprocessing.py)

Next, I wrote several methods to preprocess the data into a suitable format for a deep learning model in Tensorflow, and I additionally wrote some methods to add more anime metadata which wasn't collected during the initial web crawl, including the 'parent' anime category, which would allow anime belonging to the same series to be identified, so that an earlier anime in the same series would not be chosen for recommendation.

I removed the 'Music' category of anime series, which typically only contains short music videos and is not suitable for recommendation.

The data also needed to be separated into training and test data. The test data would additionally need to be separated into 'known' and 'unknown' data. This is because for each 'new' user, inductive models need some data on the user in order to extract their preferences and provide the prediction for the remaining unseen data.

I decided to make the first 6000 users (selected with a random seed) the training and validation set, and from the remaining users, I selected ones who had watched and rated at least 100 anime series, so that both the known and unknown sets would contain a significant amount of entries.

The number of test users ended up being slightly under 1000, with 184027 total test entries.

3 – Model Descriptions

In this section I will outline some basic theory behind the two models that will be compared.

3.1 – Inductive Matrix Factorization

Matrix Factorization is a commonly implemented collaborative filtering algorithm used for recommender systems.

The basic premise of Matrix Factorization is that the *matrix* containing the ratings for each user-item pair, R, can be *factorized* into the product of two smaller matrices. These matrices, U and V, would be representative of the users and items respectively, but would be mapped into a much lower dimensional latent space K. The goal of the algorithm then is to determine via gradient descent the best weights for U and V for which their product would best approximate R.

See Figure 1 below.

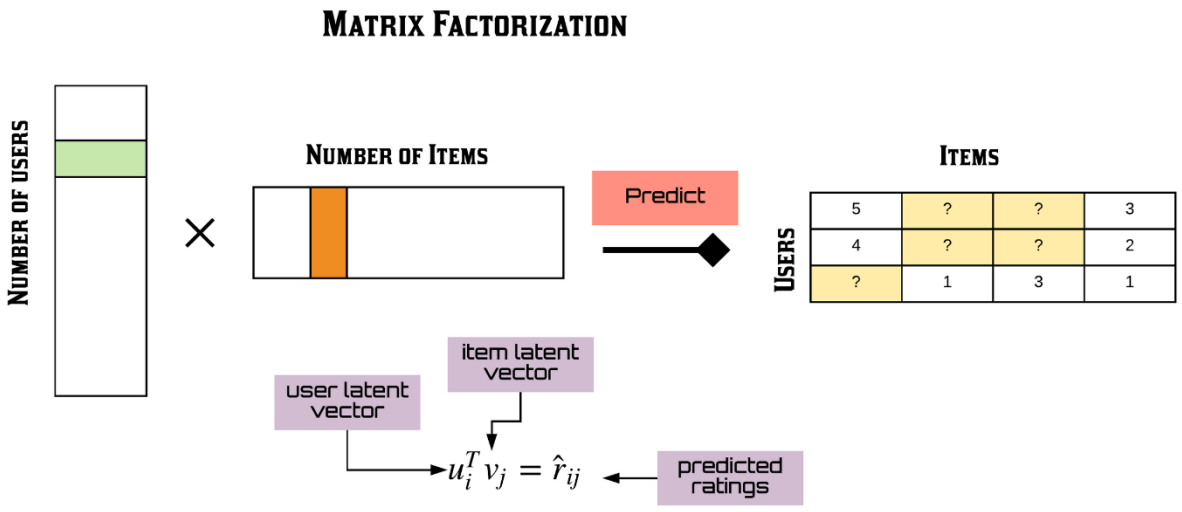

Figure 1 – Vanilla Matrix Factorization

Once the weights for the latent factor matrices are decided, the Matrix Factorization model can approximate any of the unknown user-item ratings in the ratings matrix, ie. It can 'complete' the matrix.

Vanilla Matrix Factorization can be expanded on in a number of ways, but for my base model I will just use a simple Matrix Factorization approach with bias terms. The bias terms are introduced in order to account for the fact that some users and items have higher average ratings than others. This bias needs to be written into the model.

However, as mentioned in the introduction, this algorithm cannot predict ratings for unseen users. Standard matrix factorization cannot adapt to an alteration in the ratings matrix without needing to rerun the entire model to recompute the latent matrix weights.

However, workarounds exist, one such being the One-sided Least Squares update.

The algorithm for One-sided Least Squares is shown below:

Algorithm 2: One-sided LS for incremental updates of user latent vectors p_u in MF.

Input : Training data D with triples (u, v, r_{uv}) representing user u's rating r_{uv} to item v. User u is new and item v is old. Item v has latent vector q_v, with dimension k. Number of new users m.

Result: Latent vector p_u for new user u. $u \in [1, m]$

for $u \leftarrow 1$ **to** m **do**

$p_u = (\sum_{r_{uv} \in r_{u*}} q_v q_v^T + \lambda I_k)^{-1} \sum_{r_{uv} \in r_{u*}} r_{uv} q_v$

Figure 2: One-sided Least Squares algorithm

Instead of having to recompute both the entire latent factor matrices, One-sided Least Squares allows for considerable time and memory savings via only updating the latent factors for the users if only new users are added.

Computing the latent vectors for the new users via the algorithm shown above can improve computational efficiency by a very large factor, avoiding a lot of unnecessary computation. The equivalent can, of course, be done for new items, but for my purposes I will only be incrementally adding new users.

3.2 – Inductive Graph-based Matrix Completion (IGMC)

IGMC stands for Inductive Graph-based Matrix Completion. Like Matrix Factorization, the algorithm 'completes' the user-item ratings matrix, yet employs very different methods to do so.

The first step of IGMC is to extract the *enclosing subgraph* for each user-item pair. The enclosing subgraph is a simplified representation of the relationships between the user-item pair and adjacent users and items. The enclosing subgraph extraction algorithm can be extended to neighbors multiple 'hops' away from the target user and item, but research has shown that the 1-hop variant performs very well while requiring less computation time than variants with more hops, so I shall only be using one hop in my implementation.

The enclosing subgraphs are what will be fed into the GNN (Graph Neural Network) the weights of which will be adjusted via gradient descent in order to fit the output ratings.

The core of the GNN consists of four stacked graph convolutional layers, specifically the R-GCN operator. The enclosing subgraphs are passed through the convolutional layers and then concatenated. The final representations of the target user and item from previous step are then concatenated as the pooling method. Finally, this result is passed through two dense layers with ReLu activation functions, which map the result to a scalar rating.

Figure 3 below offers a high level overview of the algorithm.

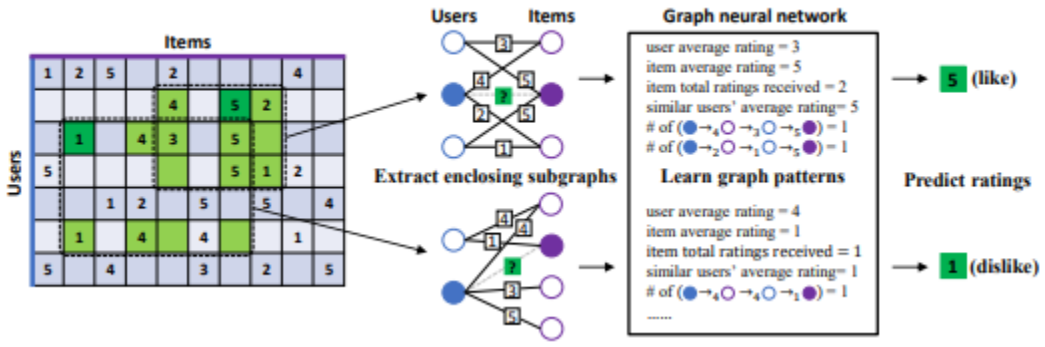

Figure 3: High level overview of the IGMC algorithm

The algorithm also employs a regularization technique called Adjacent Rating Regularization (ARR). ARR is simply a Frobenius norm between the model weights corresponding to adjacent ratings (eg. 1 and 2 or 9 and 10) so that the weights for adjacent ratings don't diverge too much. The intuition behind this is that for a more robust model, there should be some sense of 'closeness' between its interpretations of adjacent ratings, rather than each rating being interpreted as a separate entity.

Because the IGMC algorithm requires only the enclosing subgraphs as input, and not the underlying graph of all users and ratings, it works inductively. The ratings of a new user can be easily appended to the existing adjacency matrix, and the enclosing subgraphs for each of the user's item ratings can be extracted and fed into the neural network. A trained IGMC model can even potentially be used on a completely different dataset.

4 – Model Implementations (models.py)

4.1 – Inductive Matrix Factorization

I implemented my Inductive Matrix Factorization model in Tensorflow.

The unique anime and users were enumerated from 0 to their respective maximum values so that they would be compatible with Tensorflow.

The model is first initialized with the embedding dimension K, the user dimension N, the anime dimension M, and the global bias term mu, which represents the average anime rating. I created embedding layers for the users and the anime, in addition to bias terms. I then compute the dot product of the embeddings, add the bias terms, and feed the result through a Flatten layer.

The model is compiled with the SGD optimizer, and is fit on the data for the prescribed number of epochs.

The update method, which provides the model's inductive functionality, takes ratings from a new user (for already known anime) as input. The method extracts the model weights for the anime embedding layer and anime bias layer, and then applies the one-sided least squares algorithm to acquire the approximate latent vector for the new user. The method finally calculates the predictions for the user's unseen anime shows using the user's latent vector.

4.2 – IGMC

I also implemented IGMC in Tensorflow.

Muhan Zhang had already written some very clear code for extracting the enclosing subgraphs, so my code for this task was largely adapted from his, in addition to new code for creating the necessary graph structure to be compatible with the Tensorflow graph algorithm.

For the graph algorithm itself, I first initialized the four convolutional layers, and the two dense layers. The input format of the data consists of a disjoint batch of adjacency lists representing the subgraph structure.

For the call method of the graph algorithm, I separate the input adjacency matrix into a list of ten adjacency matrices, one for each possible rating. I also add the node features, which are simply indicators showing how many hops away a node is from the target node. This is the format required for stellargraph's RCGN layer, which is what I am using as the convolutional layer.

I then apply the convolutional layers to the list constructed above in a loop, appending the result of each convolution to a list called concat_states. I then implement the pooling method described in the previous section, and pass the data through the two dense layers to produce the predicted rating as the output.

5 – Model Training and Evaluation (IGMCtrain.py, MFtrain.py, and Evaluation.ipynb)

In order to train both models, I separated my user/anime/rating data into train, validation, and test sets, taking the ratings from 6000 randomly selected users as the training/validation set, and splitting this again so that 10% of this data was used to validate the models.

The test data was selected from the remaining users that had watched at least 100 anime series. This data was separated into 'known' data and 'unknown' data, the former allowing the inductive models to provide predictions for the latter.

5.1 – Inductive Matrix Factorization

I first trained the standard, non-inductive Matrix Factorization model with a number of different learning rates, in order to select the best one, which resulted in being 0.05.

I then calibrated the regularization parameter for the update method via evaluating the performance of the update method with various regularization values. I evaluated both the RMSE score and the MAP (mean average precision) score for the validation data when predicting using the update method.

A regularization parameter of 1 gave the best results. I saved the model parameters for the model with a learning rate of 0.05, ready for evaluation.

5.2 – IGMC

To implement and train the IGMC model, I wrote a custom Generator in Tensorflow, which fed the enclosing subgraph data into the model in disjoint batches. Since each batch needs to be represented as a large disjoint matrix, I kept the batch at 8 to avoid memory overflow.

I also created a function which performed a single train step, extracting the predictions and the loss (including the Adjacent Rating Regularization loss) after feeding the data into the model, and computing the gradient to update the model weights with each batch.

The main method was a loop which called the train step method for each batch generated, printing out the average loss and predictions every few hundred steps so I could see the model's progress as it ran. Training this model was a very lengthy process, and after five epochs the loss seemed to have stopped notably decreasing, so I stopped training at this point.

I saved the model weights so that the model would be ready for evaluation.

5.3 – Evaluation

I evaluated both models via 2 different metrics: RMSE and MAP. We want RMSE as small as possible, and MAP as large as possible.

I wrote a Jupyter Notebook in order to evaluate my models. I loaded the model weights and iterated through the test data, incorporating the 'known' test data in the model to predict on the 'unknown' test data for the same user.

The results are shown in the figure below. As reasonably expected, the deep learning model outperformed the matrix factorization model via both metrics, though considering the significantly greater training and computation time, the superior performance may not be enough of an incentive, as the MF model still performed very well.

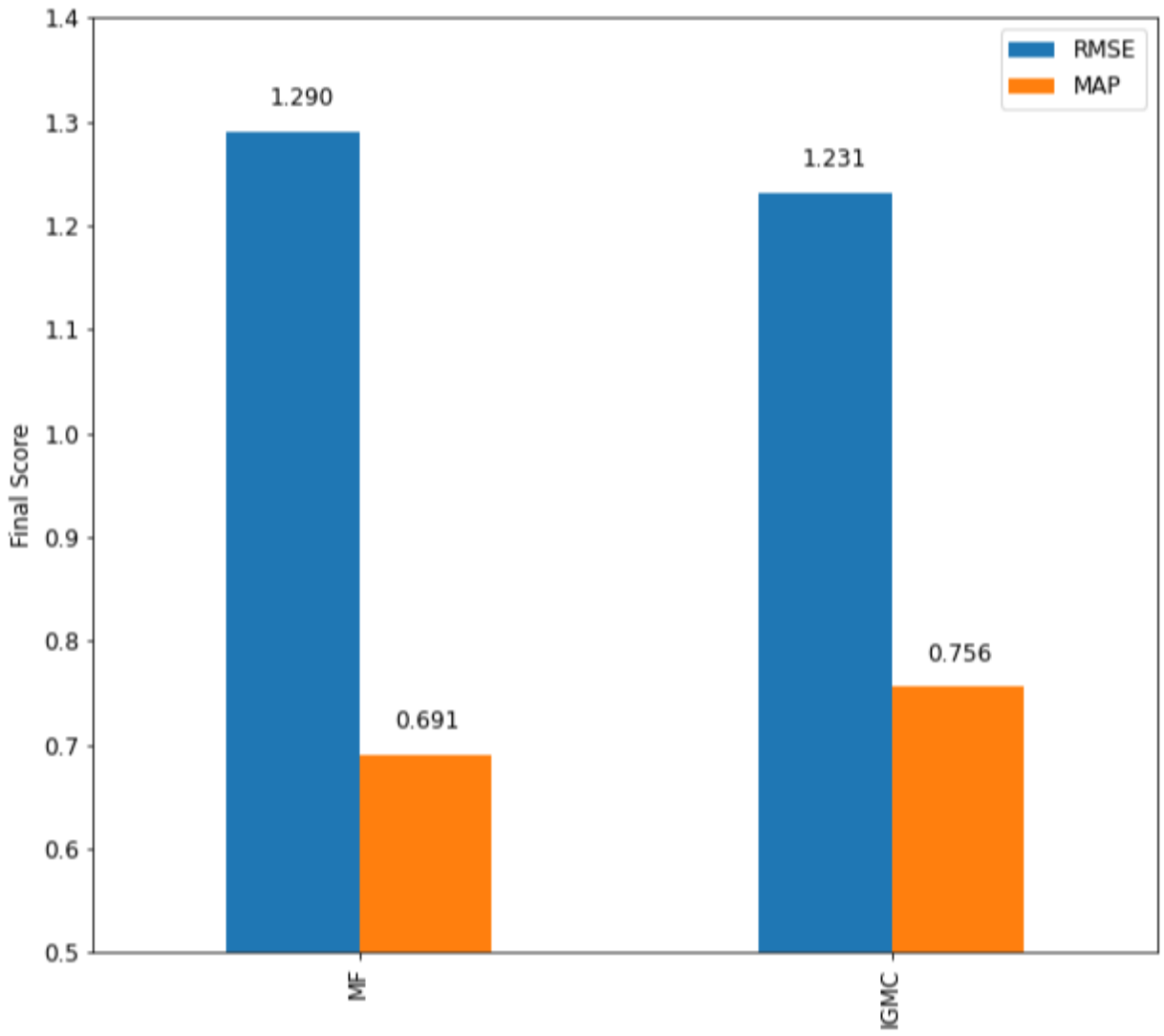

Figure 4: Evaluation Results

6 – Conclusion

Both models performed rather well in inductively making predictions on new user data. IGMC performed somewhat better than IMF, but required considerably more training time (which isn't much of an issue for inductive models since ostensibly training only needs to occur once), and also took a lot longer to deliver predictions on new users.

Therefore, deciding whether to apply the IMF or IGMC model for an application that provided anime recommendations should be conducted on a case by case basis, depending on how much of a constraint the computation time is.

Due to the long training time of the IGMC model, I did not experiment with parameters as much as I would have liked, so adjusting parameters for potentially even better results is a possible avenue of further research.