Table of Contents

Context

This project revolves around analysing the performance of different collaborative filtering algorithms. Specifically, we will be comparing the difference in performances between model-based methods, such as matrix factorization, versus memory-based methods, such as cosine similarity. To ensure a fair comparison, we will be implementing all algorithms from scratch with minimum optimisation, instead of using existing optimised implementations from popular machine learning packages such as *surprise* and *fastai*,

The dataset that is used in this project is the publicly available MovieLens 1M dataset (2003). This dataset is large and consists of 1 million ratings from 6000 users on 4000 movies. For the purpose of this project, we will be using 90% of the available ratings as the training set and the remaining 10% as the testing set. Root Mean Squared Error (RMSE) will be the evaluation metric used throughout to compare our results.

Objective

With the rise of E-commerce and streaming platforms such as Netflix, it is increasingly important to be able to implement reliable and robust recommendation systems; A good recommendation system not only improves the users' experience, it can encourage potential spending which generates revenue for the company. Consequently, it is ideal to come up with a recommendation system which is able to constantly and accurately predict what a consumer desires.

Keeping financial interests in mind, one of the key objectives naturally is to identify the best performing method. However, we understand bottlenecks and other limitations should also be considered as method practicality is just as important as performance. For instance, memory based methods have large memory requirements as recommendations are made by directly accessing the database. This trait, however, allows memory based methods to be adaptive to change, something model based methods lack.

As such, this project aims to explore the different collaborative filtering methods by evaluating both their practicality and performance.

Exploratory Data Analysis and Preprocessing

The MovieLens 1M dataset (2003) consists of a cumulative 1 million ratings of 6000 users on 4000 movies across 3 separate files, *movies.dat*, *users.dat* and *ratings.dat*. Before modelling, there is a need to examine each file in detail and account for possible errors in the data through data preprocessing.

We first check for the possibility of missing entries in the data by looking at the user and movie IDs in *movies.dat* and *users.dat*.

	MovieID	Title	Genres
3878	3948	Meet the Parents (2000)	Comedy
3879	3949	Requiem for a Dream (2000)	Drama
3880	3950	Tigerland (2000)	Drama
3881	3951	Two Family House (2000)	Drama
3882	3952	Contender, The (2000)	Drama|Thriller

	UserID	Gender	Age	Occupation	Zip-code
6035	6036	F	25	15	32603
6036	6037	F	45	1	76006
6037	6038	F	56	1	14706
6038	6039	F	45	0	01060
6039	6040	M	25	6	11106

Fig 1. The last 5 entries of *Movies.dat* (Left) and *Users.dat* (Right). Note that indexing of the table starts at 0.

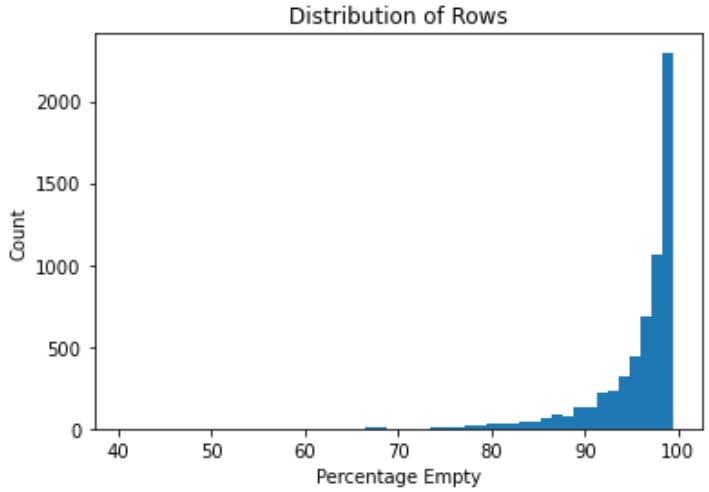
Fig 1 shows that there are no missing users but several movies are missing as the last MovieID value of 3952 is significantly higher than the length of the table, as indicated by the last index of the table (3882). Nonetheless, further EDA conducted on *reviews.dat* suggests that all reviews in the database correspond to a movie present in *movies.dat*, implying all reviews are relevant as there are no reviews written for the missing movies. As such, all reviews available in the data can be used for modelling.

Since collaborative filtering techniques are involved, there is a need to construct a user-item matrix, with each entry in the matrix corresponding to the user's rating between 1 to 5 of the respective movie. Naively, this is done by matching each review with the respective user and doing a full outer join on the movies data. This results in the matrix illustrated below.

$$\begin{array}{c} \text{Ratings} \\ user_1 \\ user_2 \\ \vdots \\ user_m \end{array} \begin{array}{c} \begin{array}{cccc} movie_1 & movie_2 & \dots & movie_n \end{array} \\ \begin{bmatrix} - & 5 & \dots & - \\ - & - & \dots & - \\ \vdots & \vdots & \ddots & \vdots \\ 1 & - & \dots & - \end{bmatrix} \end{array}$$

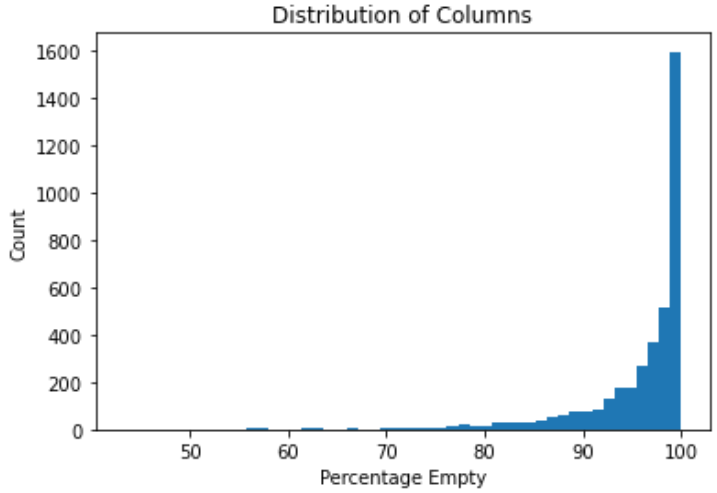
Fig 2. The general structure of a user-item matrix for the MovieLens 1M dataset

This user-item matrix for the MovieLens 1M dataset is expected to be very sparse, as there are around 4000 movies and it is reasonable to assume that most users have not watched the majority of the movies.



count	6040.000000
mean	95.735320
std	4.963869
min	40.406902
25%	94.643317
50%	97.527685
75%	98.866856
max	99.484934

Fig 3. The distribution of empty rows (users) in the item-user matrix.



count	3883.000000
mean	95.735320
std	6.281309
min	43.245033
25%	94.536424
50%	98.195364
75%	99.569536
max	100.000000

Fig 4. The distribution of empty columns (movies) in the item-user matrix

Fig 3 shows the distribution of the percentage NA of the columns present in the data. Expectedly the mean of the users, which is denoted by the 50th percentile, shows 97.5% empty rate. This means that the average reviewer has only reviewed 2.5% of the movies in the database, which is approximately 97 movies. We can also tell from the max value of 99.48% that there are no rows which are completely empty, implying every user in the database has published at least one review.

Fig 4 shows the distribution of the percentage NA of the rows present in the data. This is done to ensure we are not using movies with no prior ratings in our models, as it is not possible to give a reasonable prediction. Here, we can observe that the max value is 100%, indicating that there are indeed movies with no reviews. Thus, there may be a need to trim the data by removing these rows when running certain models.

Lastly, rating distribution of the data was analysed using the non-empty entries in the data to obtain a general sense of the spread in the data. This is important for creating a baseline of comparison, which will be elaborated in the following section on modelling.

```
The mean rating is: 3.582
The standard deviation is: 1.117
```

Fig 5. The mean and standard deviation of all ratings after preprocessing

As Fig 5 shows, the mean rating is 3.58 out of a maximum of 5, indicating that the majority of reviews are favourable. However, the standard deviation of 1.117 is rather high and this may indicate the presence of user-bias in reviews or particularly poorly received movies. Thus, we note that high variance can potentially affect our results, especially for our naive models which do not account for these patterns.

Models

There are three main methods for implementing collaborative filtering, namely: memory based methods, model based methods and deep-learning. In this project, we will be specifically exploring memory based methods that use euclidean distance and model based methods involving matrix factorization. All models implemented will be trained using the same seed of the random 90-10 train-test split of the preprocessed data. It is worth noting we will output each predicted rating as a float value instead of rounding it off to the nearest integer.

Baseline model

Two baselines were implemented based on a naive user mean and film mean model. Essentially, empty matrix entries were imputed with the mean ratings of the respective columns for the user-mean model and rows for the film mean model.

All subsequent models will be evaluated against the results of these two models, to determine if an increased model complexity has resulted in an improvement in the accuracy of prediction.

Memory Methods

Memory methods are a popular choice when it comes to collaborative filtering due to the simplicity of implementation. Here memory methods refer to collaborative filtering methods which are centred around arithmetic operations instead of parametric machine learning. Specifically, only euclidean distance is used for the purpose of similarity comparison.

Memory-based collaborative filtering can be further sub-categorised into user-based algorithms and item-based algorithms. In the context of the project, the former refers to using similar users to predict the rating for a given movie, while the latter refers to using similar movies which the user has watched to predict the rating for given movie. For either approach, a similarity matrix using euclidean distance has to be constructed to determine the pairwise similarity. Depending on the algorithm, the dimensions of this similarity matrix is either the amount of users by amount of users, or amount of items by amount of items,

From the EDA, it is noted that the item-user matrix is very sparse and over 97.5% of the matrix is empty. Thus, to avoid inducing further sparsity, item-based collaborative filtering will be

implemented as there are significantly fewer movies (3883) compared to number of users (6040). Item-based algorithms are also generally preferred as in most real-life scenarios, there are more users than items and each item tends to have more ratings than each user, so an item's average rating usually doesn't change quickly when more data is added. This leads to more stable rating distributions in the model and consequently the model doesn't have to be rebuilt as often[1].

To calculate the euclidean distance, we will be implementing two widely used approaches[2]. It is worth noting that when using memory methods, movies with ≤ 1 ratings are trimmed from the dataset. This is because we are not able to calculate the euclidean distance for similarity for these movies since there is not enough information available for logical comparisons.

Cosine Similarity

Cosine similarity is a metric used to measure the similarity of two items irrespective of their size. It measures the cosine of an angle between the two vectors of items projected in multi-dimensional space, which allows us to measure the similarity using euclidean distance.

Mathematically, the cosine of the angle of between two items is derived from their dot product divided by the product of their magnitude. Intuitively, a small angle between the vectors indicates that the items are close and therefore similarity and the converse is also true. For the case of item-based collaborative filtering, this can be calculated using the formula below.

$$Cos(\theta) = Similarity(i,j) = \frac{\sum_u^U r(u,i) \cdot r(u,j)}{\sqrt{\sum_u^U r(u,i)^2}\ \sqrt{\sum_u^U r(u,j)^2} + \epsilon}$$

Here, $r(u,i)$ and $r(u,j)$ refers to the rating for movie i and j given by the user r respectively. A small amount of noise, ϵ is added to prevent division by zero which is possible for movies with 0 reviews after the blinding of the training set during the train-test split.

After deriving the item similarity matrix, the rating of a movie for which the user has not watched can be calculated using the closest neighbours of the movies the user has watched

1 G. Adomavicius, J.J. Zhang. (2012) *Stability of Collaborative Filtering Recommendation Algorithms*. University of Minnesota

2 Sarwar, Karypis, Konstan, and Riedl. (2001). Item-Based Collaborative Filtering Recommendation Algorithms. Pg 4-6. GroupLens Research Group/Army HPC Research Centre, University of Minnesota.

based on euclidean distance and then approximated using a weighted sum. Note that our implementation for weighted sum uses all available neighbours for the ease of implementation. The formula is as follows:

$$P(u,i) = \frac{\sum_n^N \quad Similarity(n,i) \cdot r(u,i)}{\sum_n^N \quad Similarity(n,i)}$$

Where $P(u,i)$ refers to the predicted rating of user u for movie i and n refers to each neighbour of movie i .

Algorithm 1: Item-based collaborative filtering using Cosine Similarity

Derive each entry in the item similarity matrix entries using cosine similarity for each pair of items i_1 and i_2

for each unknown rating of user u and item i:

locate all neighbours of item i in the item similarity matrix

return the predicted rating using the weighted average for all neighbours

---

Adjusted Cosine Similarity

Adjusted cosine similarity functions almost identically to the vanilla cosine similarity algorithm described above. The key difference, however, is that user bias is also taken into account. In other words, some users might rate items highly in general while others might have very high standards and frequently give lower ratings. This is achieved by subtracting the average ratings for each user from his rating for the pair of items in question. The formula for adjusted cosine similarity is shown on the next page.

$$Cos(\theta) = Similarity(i,j)$$
$$= \frac{\sum_u^U [r(u,i) - \bar{r}(u)] \cdot [r(u,j) - \bar{r}(u)]}{\sqrt{\sum_u^U [r(u,i) - \bar{r}(u)]^2} \sqrt{\sum_u^U [r(u,j) - \bar{r}(u)]^2} + \epsilon}$$

Here, $r(u,i)$ and $r(u,j)$ refers to the rating for movie i and j given by the user r respectively and $\bar{r}(i)$ and $\bar{r}(j)$ are the average ratings for the movies i and j across all common users respectively. Similarly to cosine similarity, a small amount of noise, ϵ is added to the denominator to prevent division by zero.

This extra step allows us to adjust for user bias, and produces an estimation which is more sensitive towards the varying rating patterns amongst different users. The predicted rating is then calculated using the adjusted weighted average function defined below.

$$P(u,i) = \frac{\sum_n^N Similarity(n,i) \cdot r(u,i)}{\sum_n^N Similarity(n,i)} + \bar{r}(u)$$

Where $P(u,i)$ refers to the predicted rating of user u for movie i and n refers to each neighbour of movie i. and $\bar{r}(u)$ is the average rating of user u.

Algorithm 2: Item-based collaborative filtering using Adjusted Cosine Similarity

Preprocessing of matrix so adjusted similarity can be computed easily

for each rating r of user u in the user-item matrix:

subtract the mean rating of u from r

Derive each entry in the item adjusted similarity matrix entries using cosine similarity for each pair of items i_1 and i_2

for each unknown rating of user u and item i:

locate all neighbours of item i in the adjusted item similarity matrix

return the predicted rating using the weighted average for all neighbours

---

Matrix Factorization Methods

Matrix factorization algorithms work by decomposing the user-item matrix into the product of two lower dimensionality rectangular matrices As shown previously in figure 2, the movie reviews can be represented as a sparse matrix, *M*. The rows and columns of the matrix represent the *m* users and *n* movies respectively. Since there are a lot of null values, to obtain information on the missing entries, we can make use of 2 matrices, $U^{k\times m}$ and $V^{k\times n}$, where $M \approx U^T V$. U and V are known as latent factor matrices, where k is the number of latent factors. This process is known as matrix factorization or low-rank approximation.[3]

There are many proposed methods to complete the matrix using matrix factorization. We will be exploring 3 methods. Namely, Singular Value Decomposition (SVD), Probabilistic Matrix Factorization (PMF), and Non-negative Matrix Factorisation (NMF).

Singular Value Decomposition (SVD)

FunkSVD was popularised by Simon Funk during the Netflix Prize contest. To implement FunkSVD, we first initialise the latent matrices randomly. For every known rating, we increase or decrease values in the vector that correspond to the rating to minimise the error. This is essentially a gradient descent problem.

There are several well-known methods to implement loss minimization, including Alternating Least Squares (ALS), Coordinate Descent (CD) and Stochastic Gradient Descent (SGD). ALS and CD fixes one of the latent factor matrices while the other is modified, which enables parallelisation easier. SGD, however, requires the latent factor matrices to be constantly updated. For our implementation, SGD will be used due to its relative simplicity.

For FunkSVD, the prediction, $\hat{r}_{ij}$ can be calculated as follows[4].

$$\hat{r}_{ij} = \mu + b_i + c_j + {u_i}^T v_j$$

Here, μ is the global mean of all the ratings while b_i and c_j are the user and item biases respectively.

3 Reca Zadeh (2015) Databricks and Stanford. CME 323: Distributed Algorithms and Optimization, Lecture 14.

4 H.F Yu, C.J Hsieh, Si Si, and I. S. Dhillon. (2013) .*Parallel Matrix Factorization for Recommender Systems*. The University of Texas at Austin

In order to obtain the best predictions, we have to minimise the error obtained from the predicted values of the $U^T V$ and the true values of *M*. We will impose L2 regularisation on the loss function. To minimise the regularised squared error, we essentially need to minimise the following function:

$$\sum_{r_{ij} \in R_{train}} (r_{ij} - \hat{r}_{ij} \ )^2 + \lambda(b_i{}^2 + c_j{}^2 + ||u_i||^2 + ||v_j||^2)$$

Where λ is the regularisation parameter.

The minimisation step can be easily achieved by using SGD with the following update rules.

$$b_i = b_i + \gamma(e_{ij} - \lambda b_i)$$
$$c_j = c_j + \gamma(e_{ij} - \lambda b_i)$$
$$u_i = u_i + \gamma(e_{ij} \cdot v_j - \lambda u_i)$$
$$v_j = v_j + \gamma(e_{ij} \cdot u_i - \lambda v_j)$$

Where e_{ij} is the error of the current rating. The full algorithm for FunkSVD is shown below.

Algorithm 3: FunkSVD Algorithm for loss minimisation

for each iteration until convergence

for each vector *i* in *U* and *j* in *V* and $M_{ij} \neq 0$

$\hat{r}_{ij} = \mu + b_i + c_j + u_i{}^T v_j$

$e_{ij} = (r_{ij} - \hat{r}_{ij})$

$b_i = b_i + \gamma(e_{ij} - \lambda b_i)$

$c_j = c_j + \gamma(e_{ij} - \lambda b_i)$

$u_i = u_i + \gamma(e_{ij} \cdot v_j - \lambda u_i)$

$v_j = v_j + \gamma(e_{ij} \cdot u_i - \lambda v_j)$

Probabilistic Matrix Factorization (PMF)

PMF is an algorithm that is similar to FunkSVD. In order to make predictions, we will use the following formula[5].

$$\hat{r}_{ij} = u_i{}^T v_j$$

In this case, the global mean and bias terms are removed from the prediction. This means that predictions are based solely on the latent factor matrices. Similarly, we will also impose an L2 regularisation term on the loss function as follows.

$$\sum_{r_{ij} \in R_{train}} (r_{ij} - \hat{r}_{ij}\ )^2 + \lambda(||u_i||^2 + ||v_j||^2)$$

The updating steps can once again be simplified using SGD. The updates occur only to the latent vectors according to the following rules.

$$u_i = u_i + \gamma(e_{ij} \cdot v_j - \lambda u_i)$$
$$v_j = v_j + \gamma(e_{ij} \cdot u_i - \lambda v_j)$$

Algorithm 4: PMF Algorithm for loss minimisation

for each iteration until convergence

for each vector *i* in *U* and *j* in *V* and $M_{ij} \neq 0$

$\hat{r}_{ij} = u_i{}^T v_j$

$e_{ij} = (r_{ij} - \hat{r}_{ij})$

$u_i = u_i + \gamma(e_{ij} \cdot v_j - \lambda u_i)$

$v_j = v_j + \gamma(e_{ij} \cdot u_i - \lambda v_j)$

---

5 Y. Koren, R. Bell, and C. Volinsky. (2019). *Matrix Factorization Techniques for Recommender Systems*. IEEE Computer.

Limitations of using SGD

Since our preferred method for minimising the loss function was using SGD, all new vectors calculated will be required in the next iteration. This makes parallelisation incredibly challenging[6]. Our original implementation using numpy arrays will take approximately 18 hours to complete one iteration. Since 100 iterations was required until convergence, this meant the program had to run for approximately 75 days.

The main culprit for the long computational time was indexing numpy arrays. Numpy arrays are created to efficiently perform parallel operations such as matrix multiplications using vectorisation. Indexing numpy arrays always returns a copy of the array. Since our arrays are large and we have numerous calls to index the arrays, we were forced to search for more efficient means to index arrays. Hence, all our arrays will be cythonized.

Cythonization is a method of compiling python code using a C compiler. We can define our matrices to be C data types. Indexing C arrays only returns a view which makes them incredibly fast. On average, each iteration using cythonized arrays took approximately 2 seconds, which meant that our code with 100 iterations can be trained in less than 4 minutes.

6 Salakhutdinov R. and Andruh Mnih. (2018). *Probabilistic Matrix Factorization*. Advances in Neural Information Processing Systems 20.

Non-Negative Matrix Factorization

For NMF, we impose the additional restriction where the matrices we find must be non-negative. This different constraint allows the model we create to potentially be able to represent different properties of the dataset. In addition, non-negative matrices also make more sense in certain contexts where the latent factors cannot be negative.

In order to minimise the error in terms of Euclidean distance while keeping the matrices non-negative, we will be updating the matrices according to the following update rules[7].

Algorithm 5: NMF Algorithm for loss minimisation

for each iteration until convergence

$$U = U * \frac{MV^T}{UVV^T}$$

$$V = V * \frac{U^T M}{U^T UV}$$

Note that the algorithm used did not require SGD as we simply used the above multiplicative update rule to update the factored matrices U and V.

7 Daniel D. Lee, H. Sebastian Seung. (2000). *Algorithms for Non-negative Matrix Factorization.*

Results

The performances of the different methods implemented is summarised in the table below.

Model	RMSE
Baseline (User mean)	1.1818
Baseline (Film mean)	1.1426
Cosine Similarity	1.0004
Adjusted Cosine Similarity	1.0487
FunkSVD	0.8675
PMF	0.8847
NMF	0.8868

Table 1. The Root Mean Squared Error (RMSE) of the various collaborative filtering methods implemented

Judging purely on the RMSE scores, we can see that the model based methods generally outperformed the memory based methods, with FunkSVD having lowest RMSE score. This is not entirely surprising, as the user-item matrix is extremely sparse and the similarity matrix may not be a good representation of the true similarity between certain movies.

We also noted that the trained models also took significantly less time to predict a user rating compared to the memory models. As a matter of fact, predictions for model based methods were almost done instantaneously.

Baseline

The naive models implemented by imputing the mean of the rows or columns resulted in an RMSE values of 1.1818 and 1.1426 respectively. This forms a baseline of comparison for the performances of the other models. For methods which result in lower RMSE scores than the baseline, further scrutiny might be required to justify the worse performance, since all methods used are well documented.

Cosine Similarity

It is worth noting that during the train-test splitting of the data, it is possible to result in situations where a movie is not similar to any other movie due to having its only ratings blinded. This can result in memory based methods being unable to make a reasonable prediction since the similarity score will be 0. However, this is a rather rare occurrence and does not significantly affect our results.

As the results show, even a simple euclidean distance metric such as cosine similarity can result in an improvement from the baseline model.

Adjusted Cosine Similarity

Surprisingly, adjusted cosine similarity performed slightly worse than cosine similarity. This may indicate that user bias in their ratings may not be very significant by themselves. Instead, we might need to consider other factors in tandem, such as the general reception of the given movie or user preference for specific combinations of genres.

SVD

The number of latent factors can greatly affect the performance of the model. If there are too many latent factors, the model will overfit to the training data. We wanted to observe how the tuning affects the RMSE. We trained several models with varying numbers of latent factors with 100 epochs. The results are shown on the next page.

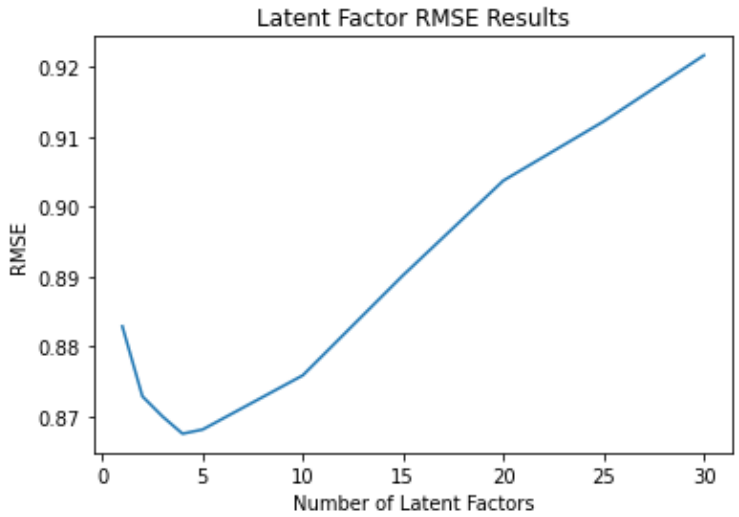


Fig 6. Plot of how different numbers of latent factors affect the SVD model

From Fig 6, we can see 4 latent factors seem to perform the best, afterwhich overfitting occurs. As such, only the score for the model trained using 4 latent factors was considered.

PMF

Similar to SVD, the greater the number of latent factors, the higher the possibility of overfitting. Hence, the PMF model was also tuned with varying numbers of latent factors with 100 epochs. The results are shown below.

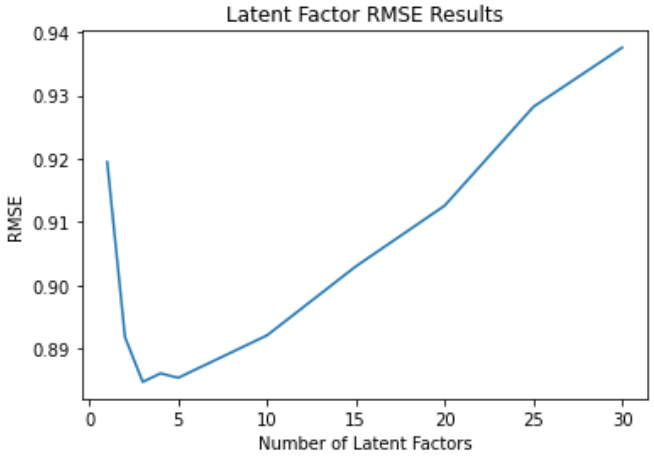


Fig 7. Plot of how different numbers of latent factors affect the PMF model.

From the results, the model started to overfit with 3 or more latent factors. The resultant RMSE for the model with 3 latent factors is 0.8847, which is slightly worse than SVD.

NMF

The trend of overfitting with more latent factors continues when using NMF. Due to using a multiplicative update rule instead of SGD, NMF takes longer to converge. Thus we used 150 epochs to train the model.

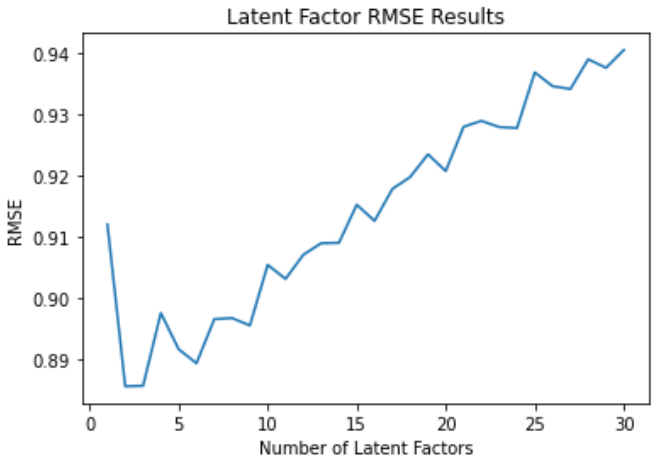


Fig 8. Plot of how different numbers of latent factors affect the NMF model

From Fig 2, we can see 2 latent factors were optimal for NMF in this case. The resultant RMSE for the model with 2 latent factors is 0.8856.

Conclusion

This project has demonstrated that both model based methods and memory based methods are viable collaborative filtering techniques. However, in cases where the user-item matrix is sparse, it is shown that model based methods, specifically matrix factorisation, outperforms memory based methods in both prediction accuracy and prediction speed.

Nonetheless, model based methods are not invariant to changes in the database and have to be retrained often to maintain performance. Since training time is the major bottleneck for such methods, they are not very pragmatic by themselves due to the high number of updates to the database every second in the real world. Another problem with model based methods is the difficulty of implementation, as there are many parameters to optimise, such as choosing the best loss function.

To build a more robust recommender system, it may be worth considering hybrid approaches, where aspects of both memory based methods and model based methods are used. Hybrid approaches essentially allow model based methods to be less sensitive to small changes in the database by incorporating additional mathematical methods. Additionally, Deep learning techniques can also be investigated as it is able to learn complex high level features which can result in better model performance, albeit at the cost of having lower interpretability.

References

1. G. Adomavicius, J.J. Zhang. (2012) *Stability of Collaborative Filtering Recommendation Algorithms.* University of Minnesota
   URL: https://dollar.biz.uiowa.edu/~street/adomavicius11.pdf

2. Sarwar, Karypis, Konstan, and Riedl. (2001). *Item-Based Collaborative Filtering Recommendation Algorithms*. GroupLens Research Group/Army HPC Research Centre, University of Minnesota.
   URL: https://asset-pdf.scinapse.io/prod/2042281163/2042281163.pdf

3. Reca Zadeh. (2015). Databricks and Stanford. CME 323: Distributed Algorithms and Optimization, Lecture 14.
   Retrieved from: https://stanford.edu/~rezab/classes/cme323/S15/

4. H.F Yu, C.J Hsieh, Si Si, and I. S. Dhillon. (2013) .*Parallel Matrix Factorization for Recommender Systems*. Department of Computer Science, The University of Texas at Austin
   URL: https://www.cs.utexas.edu/~inderjit/public_papers/kais-pmf.pdf

5. Y. Koren, R. Bell, and C. Volinsky. (2019). *Matrix Factorization Techniques for Recommender Systems*. IEEE Computer.
   URL: https://datajobs.com/data-science-repo/Recommender-Systems-%5BNetflix%5D.pdf

6. Salakhutdinov R. and Andruh Mnih. (2018). *Probabilistic Matrix Factorization*. Advances in Neural Information Processing Systems 20.
   URL: https://papers.nips.cc/paper/2007/file/d7322ed717dedf1eb4e6e52a37ea7bcd-Paper.pdf

7. Daniel D. Lee, H. Sebastian Seung. (2000). *Algorithms for Non-negative Matrix Factorization.*
   URL: https://proceedings.neurips.cc/paper/2000/file/f9d1152547c0bde01830b7e8bd60024c-Paper.pdf