ITI105 Machine Learning Project

Final Report

Team 4

21A471K - Shee Jing Le

21A466A - Ong Joo Kiat Kenneth

21A530M - Pamela Sin Hui

1. Introduction

Streaming services have been increasingly competitive with the emergence of multiple players in the industry i.e. Netflix, Disney+, Apple TV. Movie streaming services produce originals (i.e., Disney+ with the Marvel franchise) and buy rights from other film studios to create their library of films, and want to know as well as recommend movies that best suit a user's taste for them to continually watch and subscribe on the platform.

The traditional way of a business interacting with a customer is through a salesman. The salesman observes the actions, expression, gesture, emotions and any other factors or information they can get their hands on. All these with 1 goal in mind - what is this customer looking for and what can i offer him to extract the most profit out of him.

As businesses go online, the world of the salesman changes. Instead of physical interaction, it has become a virtual interaction. Instead of expression and gesture, it is now time spent and clicks. Instead of flesh and blood, it is electronical. These are all just some examples to name. Now that the customer has a different experience, how can the salesman make the best offer to extract the most profit.

Recommender systems is one of them - to give the customer the best experience.

1.1. Business objective

The goal of the business does not change - to extract the most profit.

How? - Being the best salesman it can be. The role of the salesman does not change. What changes is the salesman, the place the salesman operates, how the salesman operates and even what the salesman operates.

As we move from physical to online, the change is the platform. Now the salesman needs to know programming instead of talking. It will transform from flesh and blood to a machine. The salesman will operate the machine instead of the mouth. And it will be more reliable, applying the same methodology to each customer.

By analysing the movement across the clients, the machine will gain experience. It can be understanding of the product (content filtering), it can be user preference (collaborative filtering) or it can be a combination. From then it will be able to create an experience (give recommendations) that gives the most profit.

1.2. ML problem

Equipping the salesman with experience

The salesman over here is the machine. By providing training to the salesman, it will gain the necessary skills to recommend the correct experience. In this case, the right movie. Our team is now the salesman. We will train the salesman from 3 angles - Content filtering, Collaborative filtering memory-based, Collaborative filtering model-based.

Heuristics

Recommend a list of movies based on the best predicted rating.

Ideal outcome

To not only recommend movies that the user wants, but also to recommend movies that the user didn't know they wanted; The recommender system has to not only recommend what is obvious, but also include factors such as novelty and serendipity.

2. Dataset

Movielens dataset with movie meta-data[1]

It contains more than 260 million ratings from 260k users on movies. Data points include cast, crew, plot keywords, budget, revenue, posters, release dates, languages, production companies, countries, TMDB vote counts and vote averages. It consists of 45,000 movies' ratings on a scale of 1-5 and have been obtained from the official GroupLens website. Demographic data is missing, possibly due to privacy issues. This dataset is an ensemble of data collected from TMDB and GroupLens. The Movie Details, Credits and Keywords have been collected from the TMDB Open API.

2.1. Data analysis

2.1.1. User ratings data[1]

Both the histograms below have been truncated (in the x-axis) as there are some records which have an enormous number of ratings. There are more than 2.5k users which have more than 1 thousand ratings. It is hard to say if this is valid user data without investigating the way the data is being collected. It could also be just a difference between newer users which have been around a much shorter time than veterans. There is a large portion of the movies (more than half) which have less than 10 ratings, and some movies with up to 90k ratings. Similarly, it also just so happens that the lesser rated movies are newer. As unbalanced as the data seems, it is not unreasonable to assume that real-world data will be similar or even worse than this. And it is not really reasonable to remove users from the data as we would need to recommend movie for all users regardless of join date, and we would want to include all the movies we have in the database open for recommendations.

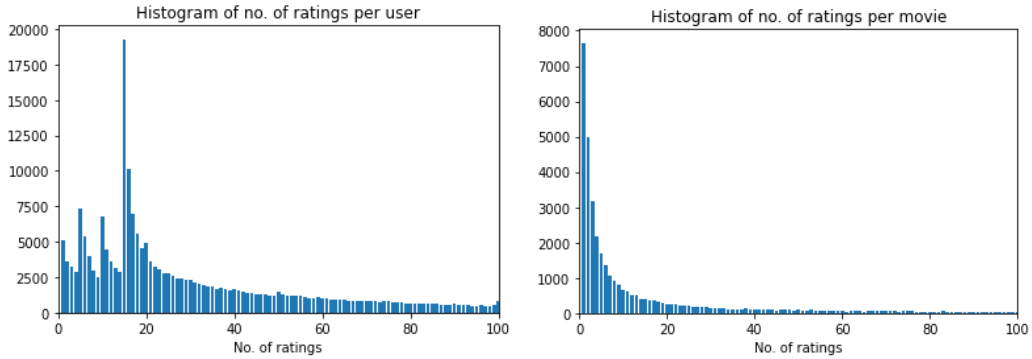


Fig.(2.1.1) Histogram of no. of ratings per user and no. of ratings per movie

The ratings of the movies, both ratings and user-mean ratings, are pretty well distributed with a mean of around 3.9. There is no evidence of under reporting as mentioned and observed in [203]. It is interesting to note that people are more likely to give ratings like 1.0, 3.0 rather than “half” ratings such as 1.5 or 3.5. It could be an issue for classification systems but since we are focusing on regression at the moment, it shouldn’t be too much of an issue. The user-standard deviation is around slightly less than 1. If a model has an RMSE of less than 1, we can be quite sure to say that the model is better than choosing the user mean. However, do note that, as will be discussed below a model that is better that predicting ratings does not mean that it is better at generating recommendations

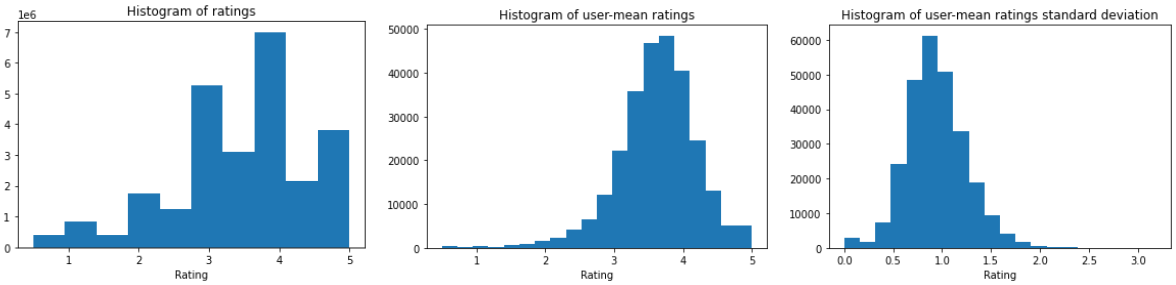


Fig.(2.1.2) Left, Histogram of ratings. Middle, Histogram of user-mean ratings. Right, Histogram of standard deviation of user-mean ratings

2.1.2 Movies meta-data

The raw files movies_metadata.csv, links.csv keywords.csv and credits.csv were first broadly examined for duplicated rows and merged together to form the data frame containing the full available information of movies. Subsequently, each of the resultant 30 raw features was examined and analysed to get the final features to be used in content-based filtering. For some raw features, a large proportion of the data belonged to a single category, making the raw feature not useful. For example, the ‘adult’ raw

feature had 99.9% of data as a boolean False. For other raw features, only a certain percentage of the data had filled values with multiple categories. For such raw features, one way we handled the data was to change into a ‘’1” for filled cells, and “0” for cells that were not filled, which seems to be an appropriate transformation. For example, the “belongs_to_collection” raw feature has multiple categories such as “James Bond Collection”, but there were data that only belonged to one collection. Broadly speaking, another technique employed for some raw features was to keep the categories where a lot of data fell into, and group smaller categories together [Appendix 101]. In summary, each raw feature was analysed individually, and a judgement call had to be made on whether the raw feature should be transformed, the kind of transformation, and whether to include in the final features to be used in the content-based recommender system.

From the enriched data frame of movie information. Each of the 30 columns of the enriched movies data frame is analysed to determine its usefulness, and useful features were subsequently cleaned and transformed to form the final features.

Column feature nam	**Analysis of raw features**	**Final features**
1. adult	99.9% of data are a Boolean False, hence we determine that this is not a useful feature to distinguish items from each other	-
2. belongs_to_collection	A few clusters of movie collections (e.g. James Bond with 26 movies in this category). Majority of filled data have a single unique value. Further, 90.7% of data are NaN. Hence, we change this data to binary 0 and 1.	90.7% - 0 9.3% - 1
3. budget	80.5% of data are 0. As this is a numeric feature, we would prefer for the majority of cells to be filled.	-
4. genres	Each movie item can have multiple genres. We find that this is a useful feature to determine the "essence" and theme of a movie.	Objective text: genres (Count vectoriser)
5. homepage	82.9% of data are NaN. Filled data are links to the movie homepage, and majority of filled data have a single unique homepage link (the top link which 12 movie items have is “http://www.georgecarlin.com”. This may be a useful feature in determining movies with high budgets that have an exclusive known homepage. Hence, we change this data to binary 0 and 1.	82.9% - 0 17.1% - 1
6. id	Not a feature - Movie item identifier	-
7. imdb_id	Not a feature - Movie item identifier	-
8. original_language	89 unique values, with "en" being the top category with 71.0% of data. The next highest is "fr" with 5.36% of data. Quite a few categories only consist of 1 movie item, and these might be very unique movies with the language in an exotic tongue. We keep categories that have at least 200 movie items, and re-categorise the remaining as "original_language_infrequent".	71.0% - "en" 5.36% - "fr" . . 3.52% - "original_language_infrequent"
9. original_title	Comparing "original_title" and "title", we find that the data is the same for 74.9% of movie items. Hence we change this data to binary 0 and 1.	25.1% - 0 74.9% - 1
10. overview	About 143 movie items have "No Overview" or NaN. Filled data is varied with some having a short sentence of 10-plus words, while others have multiple sentences.These contain insight into the plot of the movie, hence we fill cells with no real data with an empty string and utilise this as a final feature	Subjective text: overview (TF-IDF vectoriser)
11. popularity	Metric has its own method of determining popularity of the movie according to the movie database methodology.We discard this feature as we prefer to use other raw features to make a	-

	determination.	
12. poster_path	Poster webpage links that are typically unique. We decide this is not a useful feature.	-
13. production_companies	Contains production companies involved in producing the movie. This may be a useful feature as each production company may produce a certain calibre film, or produce movies that perhaps certain audiences tend to rate highly.	Objective text: production company (Count vectoriser)
14. production_countries	Contains information about which countries where the movie was produced. The top 3 categories of filled data is 'United States of America' at 39.3%, 'United Kingdom' at 4.93% and 'France' at 3.63%.	Objective text
15. release_date	Not used	-
16. revenue	83.7% of filled data are "0.0". As this is a numeric feature, we prefer if a majority of data is provided. As such, we decide not to use this raw feature.	-
17. runtime	Length of the movie item. Feature is scaled using StandardScaler.	Scaled numeric feature
18. spoken_languages	Each movie item may have multiple spoken languages. 49.3% of filled data have purely one spoken language, 'English'. Not used as we decide that this feature may have a high correlation to "original_language".	-
19. status	99.2% of data belong to the "Released" category. Not a useful feature to distinguish items from each other.	-
20. tagline	A simple few words that is a general teaser to the movie. Not used for final feature as we decide it may not be sufficient to capture things like the plot of the movie.	-
21. title	Title of the movie. For example, we find that the title "Cinderella" was used for 11 movies, which might indicate different versions and adaptations of the story.	-
22. video	99.8% of filled data are a Boolean False. Not a useful feature to distinguish items from each other	-
23. vote_average	The average vote of the movie item based on the methodology of the movie database. This takes into account factors like how many people voted, where if a large number of people voted then their collective votes have a larger impact (versus a situation where for a movie there is only one vote, despite it having the top rating). We want to make the movie recommendation for a user independent of this in-house calculated metric, and thus count on other features to capture the essence of the movie.	-
24. vote_count	The number of votes taken to derive the vote_average.	-
25. movieId	Not a feature - Movie item identifier	-
26. imdbId	Not a feature - Movie item identifier	-
27. tmdbId	Not a feature - Movie item identifier	-
28. keywords	31.6% of movies have an empty list of keywords. Some movies only have a single keyword, for example 2.82% of movies have a single 'woman director'. Other movies have many keywords, where reading the keywords gives you an idea of how the movie starts and unfolds. For example, one movie had keywords like 'fire', 'bounty hunter',	-

	'horseback riding', 'outlaw', 'unrequited love', 'pursuit', 'shot in the heart' which captures the movie plot.	
29. cast	Information like the actors and actresses that form part of the cast. The cast is the face of the movie and the audience sees the interpretation of the movie through the characters they play. We find that this is important information, for example what people generally deem as good or bad actors.	Objective text: Top 3 actors (Count vectoriser)
30. crew	Includes information like the name of the composer, editor and producer. We decide to use the name of the director as this is the person that brings the different aspects of the movie together.	Objective text: Director (Count vectoriser)

3. Evaluating recommender systems

There are mainly 3 ways to evaluate recommender systems, using past users' data (offline), conducting focus groups and lastly measuring users' response after the recommender system is launched (online). Each of them do have their pros and cons, however the scope of the project limits us to evaluating our recommender system using past users' data. Using "offline" data has its advantages in that it is reproducible, repeatable which aids in tuning of the parameters, algorithms cheaply. However, its disadvantages is that we will be unable to capture users' responses and thoughts. With only "offline" data we will have to make many assumptions into the users' preferences. And there will be many evaluation methods which we will not be able to carry out. It is also too expensive and time consuming to conduct focus groups for algorithm selection and/or hyper-parameters tuning. Therefore it is important to find a relevant evaluation metric for use with offline data.

One main objective of a recommender system in movie streaming services is to provide a positive user experience (or a more positive customer experience compared to competitors), to retain subscribers as well as to attract new subscribers. Therefore, it is important to understand what the customer wants or expects from the recommender system. As a user, it is not only important to have popular movies or other obvious choices being recommended to me. It is also important to have interesting movies, those I have never thought of before, or those which I have overlooked in the past, to be recommended to me. E.g. If I have watched the first movie of the "Harry Potter" series, it might still be ok for the next movie to be recommended to me as "Next movie to play", but I wouldn't want all of my recommendations to be cluttered up with the rest of the 7 "Harry Potter" movies. While these recommendations are "legally right", as in they are similar movies, and most probably the user will like them, these are "boring" recommendations and do not bring much value to the user experience. It will be more interesting to not only recommend movies that the user wants, but also to recommend movies that the user didn't know they wanted. These will bring us to metrics like novelty and serendipity which is covered in [2], but before these metrics can be looked at, we need to first look at the accuracy of the prediction system. Because, for these recommendations to work, the user needs to have trust in the recommender system; Trust in that the system will recommend movies that the user will like. This brings us to the evaluation metric that we will be focusing on this project: Prediction accuracy. Recommendations do not mean much if the user did not like it. It is also an evaluation metric which we can use with offline data efficiently.

Firstly, we will be using movie ratings as a proxy. Being a proxy, we assume a low rating means that the user does not like the movie, and a high rating means that the user likes the movie. However, one interesting point to take note of is that, even if the rating for a particular movie is low, the movie (or at least the trailer or the description) does pique enough interest in the user for the user to have actually watched it. Similarly, a zero in rating does not mean that the rating is beyond low; that the user does not want to watch it. It could be just that the user has not encountered this movie before or just have not watched it yet.

In measuring accuracy of our prediction, as the ratings in this dataset are split from 1 to 5 in steps of 0.5. Using MSE or MAE allows us to showcase the differences in different models/parameters even if there is only a minor difference; a classification would have bin it into one of the 10 bins.

One main difference between RMSE and MAE is that RMSE penalizes large errors more. E.g. The MAE for 2 sets of predictions with error [5,0,0,0] and error [2,1,1,1] will be the same, but the RMSE

for the first set of predictions will be bigger than the second set. Both are being used for measuring accuracy in this project. Another point to note is the difference between RMSE/MAE results per item or per user. Calculating results per user will let us understand the error a single user will likely face. However, as the distribution of ratings are not uniform/balanced through the users, the results might be overly influenced by movies by which users have fewer ratings. Alternatively, calculating results per movie will have a better representation of the prediction accuracy of our prediction algorithms. Therefore, in comparing results between different approaches and hyper-parameters, we will be using RMSE/MAE per movie.

4. Content-based filtering

4.1. Introduction

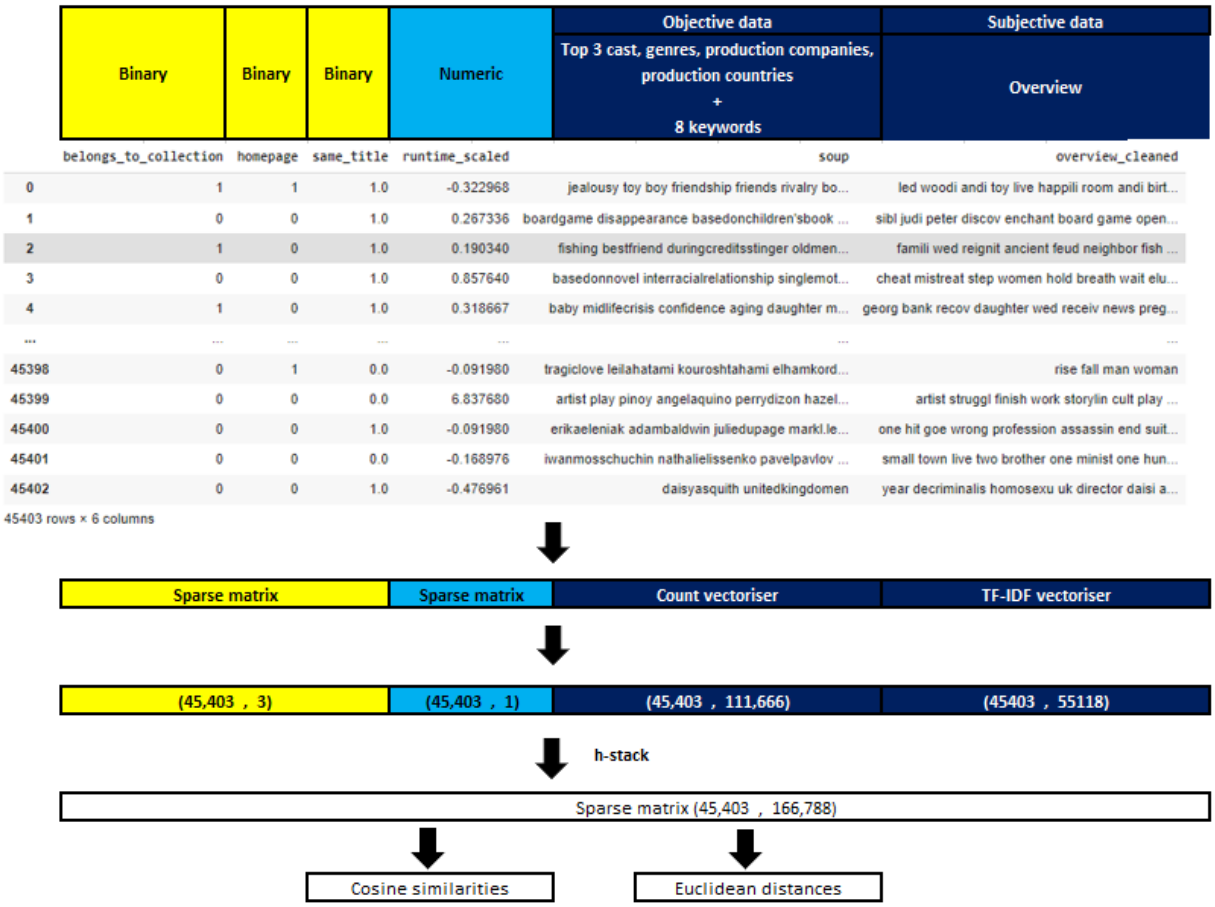
Content-based filtering leverages the features or attributes from items each user has previously rated to recommend similar items, depending on the user's previous choices heavily. An Item Profile contains information on the features of each Item. Based on the ratings each user has submitted, a User Profile is built for each user that contains information about the user's preference for each feature. Recommended items are based on item-to-item similarity, along with the user's explicit preferences [101].

This technique is more easily interpretable as we can dissect the recommendations of the method based on the features that contribute more highly to the item, where the same content can be used to explain the recommendations. This method is also less prone to the cold start problem as new items can be suggested to a user with a history, before being rated by a substantial number of other users. In addition, the use of content representations open up the use of different approaches such as text processing techniques and the encoding of semantic information as features. This type of recommender system is able to recommend items to users with unique tastes, including new items and items that are unpopular to the majority of other users [102].

However, if the features are not well-defined or if they do not distinguish between items sufficiently, recommendations will not be precise. And conversely, if there is no sufficient history of ratings for a user, recommendations will likely fall-short as we do not know the attributes liked or disliked by that user. Further, if the items already rated fall into a narrow span of similar features, there is a tendency of over-specialisation where a "filter bubble" is being created [103].

4.1. Movie item features

The final features briefly outlined in section 2.1.2 (movies meta-data) was used to build an Item Profile for each of the approximately 45,000 movies in the global movie dataset.



4.1.1. Binary and numeric data

The four non-text features are converted into a sparse matrix [Appendix 102].

4.1.2 Count vectoriser for objective data

With the availability of text data for the movies, we use Natural Language Processing methods to extract features. We create word vectors, the vectorized representation of words in a document, which carry semantic meaning [105]. The sklearn implementation of Count vectoriser converts a collection of text documents to a matrix of token counts. This implementation produces a sparse representation of the counts using scipy.sparse.csr_matrix [107]. We find that the words we have considered as “objective data” do not contain much stop words and are keywords that are not likely to be reduced further by stemming and lemmatization techniques. Hence we do not employ such techniques for objective data [Appendix 103, 104].

4.1.3. TF-IDF vectoriser for subjective data

The summary raw feature has natural written sentences and reads like typical sentences. As such, there are many stop words which do not add much useful meaning to a sentence. Stop words are removed out of the corpus of documents, and this does not sacrifice the meaning of the summary while reducing the number of resultant features derived from using Term Frequency-Inverse Document Frequency (TF-IDF) vectoriser, saving computational time [108]. The summary raw feature also undergoes the stemming normalization technique, reducing words to their word stem root form. Given that there are different grammatical forms of each word that can be used by the writer of each summary, we decide that stemming is a better alternative to lemmatization as we want to reduce the number of features and we do not require the meaningful word in the proper form. As such, through TF-IDF, we scale down

the impact of tokens that occur very frequently in a given corpus, which are less informative than features that occur in a small fraction of the training corpus [109]. In our dataset, we find that this is important as these summaries are written in

The sklearn implementation of TF-IDF vectoriser is used, where smooth_idf=True is the default. Thus, the constant "1" is added to the numerator and denominator of the idf as if an extra document was seen containing every term in the collection exactly once, which prevents zero divisions [Appendix 105]:

$$idf(t) = \log [(1 + n) / (1 + df(t))] + 1$$

4.1.4. Final movie item features

The text and non-text sparse matrices are horizontally stacked with each other [Appendix 101].One issue we encountered was that after the matrices were stacked, the stacked matrix was converted into COOrdinate format, and using this format to run cosine similarity repeatedly crashed Colaboratory due to the run exceeding the allocated RAM. The issue persisted even after paying the subscription for Colaboratory Pro, which provides higher RAM allocations. Ultimately, the fix was to convert to csr matrix.

4.2. Train and test data

The data of ratings of each user for movies the user has rated was split into 70% train and 30% test on a random state of 42. Ultimately, the measurement we want to derive is the predicted ratings for the test set, and the metric of comparison against the actual ratings in the test set is Root Mean Square Error (RMSE) and Mean Average Error (MAE).

4.3. Using 3-NN cosine similarity to derive predicted ratings and recommendations

For each user rating of a movie in the test set, the cosine similarity scores with all the movies is extracted, and subsequently only the movies in the train set for the user is filtered (we only look at the preferences for that user). The sklearn implementation of pairwise cosine similarity, or the cosine kernel, is used, computing the similarity as the normalized dot product of X and Y.

$$K(X, Y) = \langle X, Y \rangle / (||X||*||Y||)$$

An issue encountered is that the running of the cosine similarity function on the entire sparse matrix final features with itself at one go caused Colaboratory Pro to crash due to exceeding the allocated RAM. As such, a helper function to run pairwise cosine similarity on the sparse feature matrix in batches of 100 was utilised to maintain within the allocated RAM limits [Appendix 106, 107].

The cosine similarity scores are sorted in decreasing order, and the top three similarity scores are obtained. These are the three movies in the train set for that user that have the closest similarity to the test set movie in question for that user. We then find the simple average of the ratings of the three movies to derive a predicted rating. We further find the weighted average of the ratings of the three movies that is weighted against the cosine similarities, which gives a second predicted rating. After the predicted ratings are calculated, with the entire predicted ratings of each user for each movie in the test set, the RMSE and MAE is determined for each prediction type [Appendix 108].

The results show that the predicted ratings calculated through weighted average against cosine similarities is generally closer to the actual rating, and this appears to be due to the fine-tuning of the extent to which each of the three movies (in the train set for that user) are similar to the movie we want a prediction for [Appendix 104]. For example, due to the limited history of ratings in the train set we have for a user, there is not enough variety of features for that user the system has been exposed to, thus even the closest movie in the train set might not be representative of a movie in the test set.

Separately, the recommendations for a user is derived by getting the top 5 rated films from the history of that user, and thereafter getting the top two similar movies for each of these top 5 rated films with the entire movie database to provide 10 recommendations. Note that we do not employ the use of

train and test sets for these recommendations as across content based, memory based collaborative and model based collaborative filtering, we are using RMSE and MAE to draw a conclusion on performance across the three techniques. Considering future further improvements to such an implementation, the entire user history can be utilised instead by giving recommendations from the global database based on how much alike the each movie from the global database is to top rated movies explicitly rated by the user, as well as how dislike each movie from the global database is to low rated movies rated by the user.

4.4. Using 3-NN euclidean distance to derive predicted ratings and recommendations

For each user rating of a movie in the test set, the euclidean distances with all the movies is calculated, and subsequently only the movies in the train set for the user is filtered (we only look at the preferences for that user). The sklearn implementation of calculating euclidean distances, the distance matrix between each pair of vectors is calculated.

The sklearn documentation states that for efficiency reasons, the euclidean distance between a pair of row vector x and y is computed as follows [110]:

```
dist(x, y) = sqrt(dot(x, x) - 2 * dot(x, y) + dot(y, y))
```

This formulation has two advantages over other ways of computing distances. First, it is computationally efficient when dealing with sparse data. Second, if one argument varies but the other remains unchanged, then dot(x, x) and/or dot(y, y) can be pre-computed. The sklearn documentation further points out that this is not the most precise way of doing this computation, as this equation potentially suffers from “catastrophic cancellation”. Also, the distance matrix returned by this function may not be exactly symmetric as required by, e.g., scipy.spatial.distance functions. In consideration of the RAM limits provided by Colab Pro, we utilise the sklearn implementation to compute euclidean distances.

The euclidean distances are sorted in increasing order, and the smallest three distances are obtained. These are the three movies in the train set for that user that have the smallest distance between the test set movie in question for that user. We then find the simple average of the ratings of the three movies to derive a predicted rating. We further find the weighted average of the ratings of the three movies that is weighted against the inverse of the euclidean distances (distance is inversely proportional to similarity), which gives a second predicted rating. After the predicted ratings are calculated, with the entire predicted ratings of each user for each movie in the test set, the RMSE and MAE is determined for each prediction type [Appendix 109].

Similar to the results in section 4.3 using cosine similarities, the results show that the weighted average is generally closer to the actual rating, and this appears to be due to the fine-tuning of the extent to which each of the three movies (in the train set for that user) are similar to the movie we want a prediction for [Appendix 110].

Separately, recommendations to each user is calculated in a similar fashion to that laid out in section 4.3.

4.5. Results

	Content-based (euclidean distance,3-NN, simple average)	Content-based (euclidean distance,3-NN, weighted average)	Content-based (cosine similarity,3-NN, simple average)	Content-based (cosine similarity,3-NN, weighted average)
RMSE	1.0642	1.0623	0.9856	0.9816
MAE	0.8188	0.8172	0.7471	0.7442
Test set	30,327 predictions (~1000 users in test set)		30,327 predictions (~1000 users in test set)	
Time	Approximately 4.5 hours, high-ram setting on Colab Pro		Approximately 4.5 hours, high-ram setting on Colab Pro	

From the results, among the four implementations, content-based 3-NN cosine similarity using weighted average performed the best with an RMSE of 0.9816 and MAE of 0.7442, while content-based 3-NN euclidean distance using simple average performed the worst with and RMSE of 1.0642 and MAE of 0.8188. A reason for better ratings predictions using cosine similarity as opposed to euclidean distance is, for example, that one of the final features, “summary”, have texts of very different lengths, and using cosine similarity even if words have broadly different counts but appear in text in the same proportion, that proportional relationship is factored in and not the counts.

We see that there is a disparity in information on movies, where some movies have an extension list of keywords tagged to it and a detailed overview being written, while other movies may have a lack of objective and subjective text and categories linked to them. The computed similarity scores and distances of the movie items are dependent on the available information about the movies at our disposal. Certain gaps in the data have an extensive impact on similarity or distance measurement of movies with each other, thus affecting the predicted movie rating (and subsequent RMSE and MAE metrics) and bad movie recommendations may form part of the output.

The results of content based filtering caused us to reflect on whether the raw features provided in the dataset was even sufficient to encode what makes a movie, and what aspects of movie items can propel users to like or dislike films. A movie is composed of many elements strung together, from the plot, story arch, dialogue, acting, character development, music, cinematography and lighting. From a layman’s perspective, the features in our dataset such as the names of the director, actor, name of production company, length of film and summary of the movie, may not even come close to capturing the makeup of a movie. A movie engages audiences mentally, from the timing of the script, specially curated scenes, music accompaniment and actors’ facial expressions. There are multiple layers and aspects to movies that are extremely difficult to encode in features, and our dataset is missing elements that truly makes an entire movie. For example, while a well-known director may have directed a movie well-liked by many users in the past, it is not a given that the director performs as well in another movie project. From this perspective of thinking along the lines of whether our features sufficiently represent movie items, we conclude that our dataset is missing elements of the makeup of movie items. As such, in this instance, the use of content based recommender methods is not ideal, and similarity scores and distances between movie items may not be ideal, accounting for this method not performing as well in this implementation using this dataset.

Apart from the discussion of RMSE and MAE of predicted ratings above, the movie recommendations for each user was also run at random to see if any patterns emerge. A few movies tend to be commonly recommended across users such as 'The Drunk' and 'Superpower'. A look at the features of these movies show that these commonly recommended movies appear to have missing information like the cast names, director’s name, genre and production company names. As such, the objective data final feature is extremely short. The lack of other words and the appearance of common words such as the production country of “unitedstatesofamerica” is perhaps what leads to a resultant close similarity with other movies, hence skewing the list of recommendations for a user to certain movies. This observation reiterates the effect of the disparity in information on features across movie items, where insufficient data makes the calculation of similarities and differences across items incomplete. We can workaround this issue by removing movie items that we deem not to have sufficient data for from the global database, forming indicators of what constitutes “sufficient data”.

In considering the scope for future exploration and project enhancement for content based recommenders, further enriching text embedding usings pre-trained word vectors such as Word2Vec is a promising avenue, to provide greater semantic meaning to subjective text data instead of simply considering the word occurrence in count vectoriser and TF-IDF. Although this may require considerably more computing power, we find that this is a worthwhile avenue to pursue as it is a way to dissect the words and give it a meaningful representation. Going a step further, Bidirectional Encoder Representations from Transformers (BERT) is one of the latest techniques to try in further delving into NLP techniques.

5. Collaborative filtering

Collaborative filtering makes predictions based on other users' data in contrast to content-based filtering where predictions are based on the movies' metadata (with the only user data being the user's data itself). CF could generate recommendations which have higher serendipity than content-based filtering as they are not limited to movies' metadata in making decisions. The recommendations from CF come from other similar users/items which could have surprising results. I.e Each user in an individual which could have watched or have known very different movies from another individual even if they have perfectly similar tastes.

Another advantage of CF is that it is independent of the information about the users or movies, which removes the dependency on the method in which the movie is being tagged or categorized, which could be biased.

5.1 Collaborative filtering memory-based

By memory-based, it is non-parametric and is based on past-data.The kNN method is the most common memory-based method that is used. However, it is too computational intensive and we will be using an alternative method, the approximate nearest neighbour method for "nearest" neighbour searching.

5.1.1. k-Nearest Neighbours

In CF recommender systems, there are 2 types of neighbours[202], namely "user"-neighbours, a.k.a. User-based CF and "item"-neighbours, a.k.a Item-based CF.

The motivation behind "user"-neighbours or user-based CF is that if user A have similar tastes (rated movies similarly) as user B, a movie which is highly rated by user A will most likely be highly rated by user B. Similarly, the motivation behind "item"-neighbours is that if many users have rated movie A and movie B similarly, user X will most likely rate movie B similarly to movie A.

While simple in concept, there are a few challenges in using kNN methods in real-life situations especially where there is a huge amount of data. Being a memory-based method, the amount of space/memory needed to store the data and the computation time needed at prediction could increase very quickly. Even though there are techniques used to reduce computational time, i.e. KD-trees, they do have their own limitations. KD-trees are further limited by the "no. of data points" vs "no. of dimensions". In the case of user-based kNN, our data set has a dimension (movies) of around 45k and a user-base of around 270k users, which is a ratio of 1:6. There is a high chance where many of the nodes are just with one leaf, and even if the nodes are possible, the depth of the tree will be very shallow and the computational cost saving of the algorithm will be severely limited. In the case of item-based kNN, a dimension of 270k vs 45k. Most of the nodes in the tree do not even have any data to split.

5.1.2. Approximate Nearest Neighbours

There are many cases where the exact nearest neighbours need not to be found, or rather, might not be worth the extra computational time that is needed. Approximate nearest neighbours increases speed at the expense of accuracy. This can be very useful in applications where accuracy is not as important as the speed at which predictions can be made. In our case, the relationships between users and items (e.g. similarity of users based on common movie ratings) are already fundamentally weak. Users with common movie ratings does not necessarily mean they will have similar taste in all other movies. Likewise, users with similar tastes do not necessarily have common movie ratings. The calculated similarity might not represent the "actual" similarities between users fully. Therefore, the loss of accuracy in the exact nearest neighbours might not be too much of a concern.

We will use LSH with random projections[201] where "n" number of random hyperplanes are generated and the data points are separated into buckets depending on the location of the data points with respect to the hyperplanes. Thereafter, to find the nearest neighbours for a new data point, the bucket for which the new datapoint belongs to is calculated, and then the data points which belong in that bucket would be the candidates for the nearest neighbours. For each of the candidates, the distance would be calculated and the top "k" candidates would be the approximate nearest neighbours for the data

point.

5.1.3. Distance metrics

In the context of a recommender system, cosine distance can be very useful. It can capture if 2 users have bought/listened to the same thing, no matter the number of times it was bought/listened to. However, in this dataset that we are using, the “length” of the vector is actually dependent on the ratings of the movies. E.g. 2 users who have rated 2 movies, the first user rated both movies 5 (she/he really likes them alot), the second user rated both movies 1 (the movies really aren’t her/his cup of tea). Logically, these two users should not be similar (at least in the context of movie preferences); they should even be opposite of each other (cosine distance cannot be negative). However, if we calculate the cosine distance between them, the distance is actually 0 (cosine similarity = 1, they are as similar as it can get).While this might seem to be like a special case, it is still important to take note that cosine distance/similarity does not take into account the mean and variance of the users’ ratings.

$$CV(u,v) = \cos(\mathbf{x}_u, \mathbf{x}_v) = \frac{\sum_{i \in \mathcal{I}_{uv}} r_{ui} r_{vi}}{\sqrt{\sum_{i \in \mathcal{I}_u} r_{ui}^2 \sum_{j \in \mathcal{I}_v} r_{vj}^2}},$$

Cosine distance formulae

While Pearson correlation[202] addresses mean and variance of the ratings, it does not use the “true” user-mean, it only uses the mean of the items which are rated by both users. This could be an issue if the user-item interaction matrix is sparse. There can also be an issue when we are dealing with item-based interaction. when Pearson correlation is calculated for item-based similarity, the mean of the items will therefore be the item-mean instead of the user-mean. Adjusted cosine similarity is where the “true” user-mean is being used not only in user-based similarity, but also in item-based similarity. Therefore, this could address the issues with cosine similarity as well as Pearson correlation and should fit our dataset the best.

$$AC(i,j) = \frac{\sum_{u \in \mathcal{U}_{ij}} (r_{ui} - \overline{r}_u)(r_{uj} - \overline{r}_u)}{\sqrt{\sum_{u \in \mathcal{U}_{ij}} (r_{ui} - \overline{r}_u)^2 \sum_{u \in \mathcal{U}_{ij}} (r_{uj} - \overline{r}_u)^2}}.$$

Adj. cosine distance formulae

One major difference between cosine distance and adjusted cosine distance is that since adjusted cosine distance is the difference with the user-mean, there can be negative values, the distance could be negative. In the previous example, instead of cosine distance = 1, we will end up with a negative adjusted cosine distance (if the user-mean is known). E.g. with both users’ mean =3, ratings of [5,5] will be [2,2] and ratings of [1,1] will be [-2,-2], the adjusted cosine distance will therefore be -1. Logically more correct as discussed; The adjusted cosine distance actually penalises for ratings which are different, i.e. it penalises the distance metric if one rates as good vs the other one who rates as bad, unlike cosine distance will both improves the distance, albeit at different rates, as long as there’s a similar movie rated. Fig.(1) shows how the profile of user-pairwise distance (for a single user) is changed with the change in distance metrics. Both of them have a large portion of the users which have distance = 1, orthogonal. This is to be expected from a high dimensional sparse matrix. From the perspective of our dataset, in order for two vectors (users) to be non-orthogonal, they must have at least one common movie which is rated. With a set of more than 45k movies, there could be plenty of chances that 2 users will not have rated the same movie at all. Therefore, due to this high dimensionality, not only can we observe that there is a large portion of orthogonal users, the distance between users is quite large in general. However, it is important to note that the chances of a movie being rated is not uniform across the set of movies.

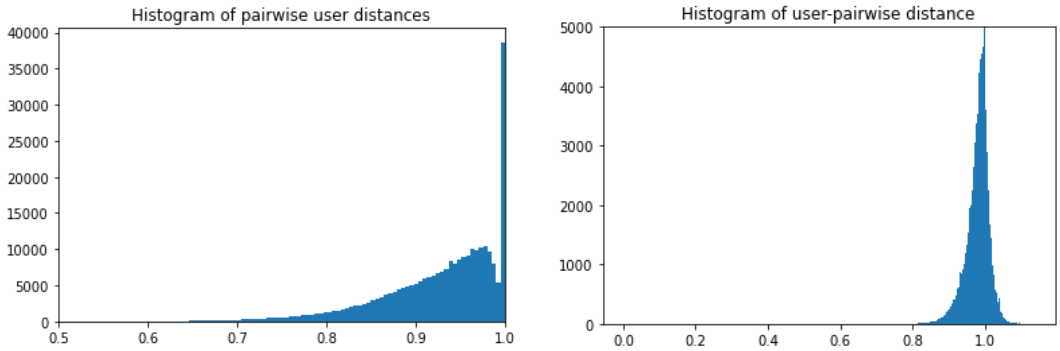


Fig.(2) Histogram of pairwise user (for user id: 99427)

5.1.4 Empirical comparison of kNN vs aNN

To compare the accuracy/efficiency of the approximate neighbour method, we will mainly look at the 1 criteria: That would be that the nearest neighbours that the approximate neighbour method found have to be close enough to the actual nearest neighbours. It will be compared to a group of users which are drawn randomly as a heuristic. The LSH method used here will be using 10 random projections and 1 hash table. It will be done on both unscaled dataset as well as user-mean scaled dataset (adjusted cosine distance).

All pairwise distances for the user-id 99427 will be calculated by brute force to serve as “ground truth”. Next, candidates (from the bucket list from the LSH method) are selected. Only one set of candidates from the LSH method will be generated for the unscaled dataset, but multiple sets of candidates which correspond to different numbers of projections and different numbers of hashtables.

The actual ranking (in reference to the rankings calculated by brute force) for the top 10 users from both candidates from the LSH method and the random selected users were calculated, alongside with the difference in distance. The nearest approximate neighbour found by the LSH method has ranked in the 20s up to even the 10s. Considering that we have over 45k users, it might not be a bad approximation. In contrast, the nearest neighbour found from a group of randomly selected users is ranked around 300s and 500s. Also interesting to note, there are also candidates in the LSH method where the distance = 1, orthogonal to the queried user’s vector. It is unsure if this is commonplace in LSH methods, but these users will be filtered and removed from the list of “valid” candidates for prediction.

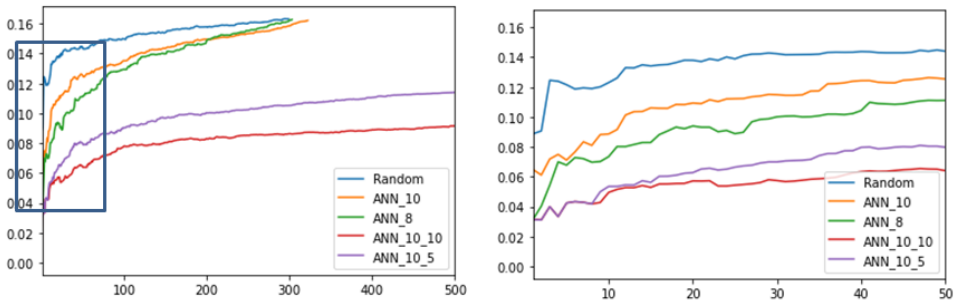


Fig.(2)Plot of distance differences between similarly ranked users.

In this experiment, the approximation made by the LSH method certainly seemed to be an improvement over randomly selected users. However, with only 1 hashtable, this only holds for the first few approximations. As we can see from both the rankings and the plot of distance differences, the results from the approximation by the LSH method quickly converge with the results from the set of randomly selected users. The usage of multiple hash tables addresses this issue as it provides not only a larger pool of candidates but also candidates from hash tables which are randomly different from each other (which therefore reduces the total error). However, each additional use of hash tables increases the

computational time. In order for us to use nearest neighbours to predict a user's movie ratings, we need more than a few neighbours, therefore it would be best to make use of multiple hash tables. It is simply because there will be no pair of users that have watched all the same movies. To predict ratings for multiple movies, we need to find the nearest neighbour who has rated each particular movie. These neighbours are not necessarily in the top 5 or top 10 users. This is apparent with the following experiment with the unscaled dataset. There are a total of 163 movies that ratings need to be predicted for user id 99427. While predicting the ratings, the number of movies whose ratings are not found (which means nobody has rated the movie in the list of neighbours) are counted. It is carried out with different numbers of neighbours and the results are tabulated. It can be seen that the nearest neighbours found by brute force have most of the movie ratings found with k=50, while the nearest neighbours found by LSH and random selected users do not have all the movie ratings found even with k=417 (which is the number of valid candidates in the LSH bucket). This is actually very interesting as it really showcases the core principle behind user-based collaborative filtering; similar users have similar taste in movies.

	kNN	aNN	random
K = 10	14	36	54
K = 50	1	25	28
K =100	0	21	23
K = 417	0	17	16

Table.(1). kNN vs. aNN, number of movie ratings not found

On the side note, using the approximate nearest neighbour method will most likely generate a different set of recommendations as compared to kNN. The impact of this difference can not be studied offline and not certainly in this project.

5.1.5. Implementation

Data handling

This dataset consists of around 26 million ratings from more than 270k users, spread over more than 45k movies. Each user has been made sure to have at least 20 ratings. Assuming uniform distribution, this averages out to be about 100 ratings per user, with each movie rated about 600 times. The size of the user-item interaction matrix would then be 270k by 45k, the user similarity matrix (used in Surprise library) would then be 270k by 270k. With 26 million ratings, the density of non-zeros in the user-item interaction matrix would be about only 0.2%

Both numpy arrays and pandas dataframe are “in-memory”. Pandas (“bugfix” is supposed to be released in the developer releases) do not support, as of version 1.2.4, dataframes with more than int32 number of elements (i.e. 4.3billion elements. We have about 12.1 billion elements in the user-item interaction matrix).

As mentioned, one thing about numpy arrays and dataframes is that they work “on-memory”. This naturally led me to look for “on-disk” methods. Not only do they address the issue of insufficient memory, they also provide a kind of persistent data storage. One of the methods is the library h5py which utilises hdf5 format files. The dataset which is stored in the hdf5 can be accessed with an array-like syntax and could be read in multi-process in parallel. Most importantly, slicing is efficient. As we will need to do plenty of slicing operations while calculating similarities and such, slicing performance is important to us. A variety of other data persistence methods is being employed in this part of the project. Redis is being used to store the LSH data (hashes of all the data in the training set). Redis is used just because the library supports it and methods are already provided. A python pickle is being used for the index-id dictionary that is used in tandem with the hdf5 files.

Multi processing

Python code typically only utilizes a single core most of the time. To make use of the multi-core technology of modern CPUs, there is a multiprocessing library that we can make use of. This library basically splits the evaluations into a number of processes which will be run in parallel.

Special libraries

Some “special” (not introduced in the course) libraries will be used in this part of the project. One will be the h5py library[reference to homepage] to employ the usage of hdf5 files for “on-disk” operations as well as persistence of data. The other library will be the NearPy library to employ LSH with random projections.

Evaluation metrics

As mentioned in the section, 1.3. “Evaluation of recommender systems”, it is difficult to assess a user's reaction to recommendations and preferences just using an offline history of ratings. Using the user’s ratings as a proxy for user preference, the best evaluation method of the model would be a comparison between a predicted rating and the actual rating of movies by each user based on RMSE and MAE on a regression model. However, because of it being a regression, we need to take note that with a large enough number of neighbours used, the regression results will tend to converge into the sample mean, which might not be meaningful. In contrast to classification based models, where it will always bin to one of the values based on majority votes. E.g. The average of ratings 1 and 5 will be 3 (results based on regression), but classification will only bin it to either 1 or 5, depending on the probabilities. This will not be further explored in this project. In cases where movie ratings are not found in the nearest neighbours, we will leave the rating as 0. Because if no neighbours have watched this movie, we couldn't have recommended it anyway. If the "ground truth" is that the user will rate it a five, we will have the maximum penalty from missing a recommendation which the user will really like; if the "ground truth" is a one rating, we will still have penalty, but minimal, from the fact that the user have actually watched it before rating it one.

Data scaling

Even though the ratings range from 1 to 5, not every individual will use the full range of ratings. Due to cultural/societal influences [203], there will be some differences in how people rate movies. some might rate most of the movies highly. Even the worst movie they have seen might score a 3. In contrast to a person who is very critical, giving most of the movies a 1 or 2, with the best movie scoring at most 4. This gives rise to significant variance between users. As mentioned in the distance metrics, user-based mean differences will be used in this project to better represent each rating across different users[202]. This basically converts cosine distance calculation to one of adjusted cosine distance.

Heuristic

Heuristic used in this part of the project will be one of neighbours randomly selected instead of the LSH method.

5.1.6 Results

Firstly, the unscaled dataset will be run with K = inf, which means that all of the candidates (less those distance >=1) in the bucket will be used. This will show us the maximum percentage of movie’s ratings that can be found in the group of candidates. As discussed earlier, due to the “approximate” nature of the current method, there will be movie ratings that need to be predicted but are not found within the group of candidates. This should provide some sort of a metric by which we can compare results with; the lesser the portion of ratings not found, the closer is the “approximate” to the nearest neighbours. As also previously discussed, one feature of a regression model is that the more neighbours used, the closer the result will be that of using the data-mean itself. This is something to note; even if as K becomes larger, the smaller the results it gets (our current RMSE/MAE is larger than that generated from using the mean), it does not mean that it is a better model for recommendations. Therefore, results will be compared on the basis of fixed K (if the bucket of candidates allow). K=50 and K=100 is chosen based empirically on the earlier graph before it becomes linear.. As discussed, K cannot be too small as well.

Due to time and memory limitations, the experiments will be carried out based on sampling, and not all experiments can be carried out on the same number of samples. To compare results which are close together, it would be best to compare to those which are carried out on exactly the same samples

(shaded in different grey).

User-based CF				
	Unscaled. K = inf	Scaled. K = 50	Scaled. K =100	Scaled. K = inf
Projections: 10 Hash Tables: 1	2.35/1.81, 37% 1.36/0.90	2.85/2.35, 55% 1.14/0.90	2.73/2.20, 48% 1.12/0.90	2.60/2.03, 39% 1.14/0.92
Projections: 10 Hash Tables: 5	-	3.13/2.70, 69% 1.04/0.79	2.56/2.02, 48% 0.95/0.73	-
Projections: 10 Hash Tables: 10	-	2.88/2.40, 60% 0.98/0.75	2.34/1.81, 39% 1.06/0.80	-
Projections: 8 Hash Tables: 1	-	2.74/2.22, 55% 0.99/0.78	2.61/2.07, 48% 1.03/0.79	-

Item-based CF				
	Unscaled. K = inf	Scaled. K = 50	Scaled. K =100	Scaled. K = inf
Projections: 10 Hash Tables: 1	3.02/2.61, 69% 1.21/0.90,	3.30/3.04, 88% 1.15/0.86	3.30/3.04, 88% 1.15/0.86	-
Projections: 8 Hash Tables: 1	-	2.83/2.38, 60% 1.23/0.92	2.83/2.38, 59% 1.23/0.92	-
Projections: 8 Hash Tables: 5	-	2.18/1.59, 33% 1.01/0.74	-	-
Projections: 8 Hash Tables: 10	-	2.06/1.49, 28% 1.08/0.79	-	-
Projections: 6 Hash Tables: 1	-	2.36/1.80, 39% 1.10/0.80	2.05/1.47, 26% 1.18/0.79	-

Table.(2) Hyper parameters tuning

First we take a look at the difference between unscaled and scaled datasets (i.e. cosine distance vs adj. cosine distance). The difference in results in user-based CF is more pronounced than in item-based CF. This could be due to the fact that in user-based CF, prediction of the rating is done by averaging the ratings across the neighbours, where the effects on the scaling could be much more pronounced as compared to item-based CF, where the prediction of the rating is done by averaging the ratings across similar movies within the user. While the adjusted cosine distance should better describe the similarities between the movies, since we are using just “approximates”, the effects of the adjusted cosine distance might be buried.

Next we will take a look at the effects of decreasing the number of projections; increasing the bucket size. The effect of this change is much more pronounced in item-based CF than in user-based CF. It could just be that the starting point, projections =10 is too little for item-based CF (item-based CF has much less data points than user-based CF, 45k vs 270k). As we can observe from the very high portion of ratings predictions which ratings are not found. Or it could just be from the size of the sample, which is not too affected by the reduction of 2 projections (2bits in the bucket). The projections are completely random, which can also the inconsistent effect in the results can be attributed, especially with a small sample size.

However, by increasing the number of hash tables, the results improved. The portion of unfound ratings decreased across the board and while the RMSE/MAE results are mixed, it can be attributed to different sample sizes. The drawback is that with an increasing number of hash tables, the memory usage and the computational time increase proportionately. The cost-benefit balance remains to be investigated.

Lastly, the effect of K, the number of nearest neighbours is to be looked at. While in general,

K=100 reduces the portion of unfound ratings, there are some cases, (even in experiments with the same samples), the RMSE/MAE increases. It can be attributed to the fact that with the reduction of the portion of unfound ratings, more ratings are actually counted towards the RMSE/MAE. However, from another perspective, we can say that the “new” ratings prediction are not as accurate and it implies that even though the ratings are found, the neighbours are too far away to have an accurate (and meaningful) enough prediction.

In view of the above discussion, one set of parameters each for user-based CF and item-based CF will be used to run on a test set.

	KNN Scaled, K=100	Item-based, Scaled Proj.=8, Tables=5	User-based, Scaled Proj.=10, Tables=5
Results	1.04/0.77	1.03/0.76	0.97/0.75

Table.(3) Final results

5.1.7 Recommendations generation

Recommendations will be generated separately for user-based and item-based. Item-based recommendations are more easily interpreted, as seen in e.g. Netflix “Because you have watched xxxx”. It can also be easily combined/worked together in hybrids with content-based recommendations (since both deal with similarities in movies). Being interpretable generally improves the user experience and increases the trust the user has in the system. User-based recommendations are usually more privacy sensitive and will be less interpretable for the user. The recommendations could be stated as e.g. Spotify’s “Discover Weekly” etc.

Recommendations will be generated from the list of highest ratings predicted from top K neighbours. Ratings which are generated from only 1 neighbour will be discarded (can be tuned, however 1 is used in this project as there will be no way to do a quantitative evaluation at this point in time). The reliability/confidence of the recommendation from only 1 neighbour can be disputed. To increase the serendipity of the recommendations, very popular movies are also excluded (i.e. movies generated from >20 neighbours). A few random selection of the ratings predicted could also be included to spice things up but it is not implemented in this project.

5.1.8 Future work

A deeper exploration of LSH techniques to improve performance, e.g. data-dependent hashing techniques. Optimization of the code to increase efficiency, e.g. reduction of redundant calculations, usage of numpy arrays instead of dataframes etc.

5.2. Collaborative filtering model-based

Intro to collaborative filtering – model based.

Collaborative Filtering – Model Based looks at the user-item interactions where users and items representations have to be learned from interactions matrix. Assumes a latent interactions model that needs to learn both users and items representations from scratch. High bias, low variance.

Model based collaborative approaches only rely on user-item interactions information and assume a latent model supposed to explain these interactions. For example, matrix factorisation algorithms consists in decomposing the huge and sparse user-item interaction matrix into a product of two smaller and dense matrices: a user-factor matrix (containing users representations) that multiplies a factor-item matrix (containing items representations).

5.2.1 Surprise library:

Data:

Using the surprise library for python does not require much data preparation. The needed data is user, movie and rating. For the smaller dataset, 100,000 ratings by 943 users on 1682 items, each user has rated at least 20 movies. For the bigger dataset, 1,000,000 ratings, 6040 users, on 3952 items, ratings are made on a 5-star scale (whole-star ratings only), each user has at least 20 ratings. Our whole dataset has 26,024,289 ratings, 270,896 users and 45,115 movies.

Method/preliminary experiments:

The first attempt was to run the codes using the whole dataset and it took too much time. So I stopped the process and did it in increasing samples hoping to find a peak based on the lowest RMSE, the metrics chosen to evaluate the model. The results was consistent with what was suggested. Being a non-parametric, unsupervised model, the bigger the dataset, the lower the RMSE and thus the better the algorithm performed. [see appendix 301]

The second attempt was the comparison across models. Surprise library has multiple input models. Also, since this is only for model comparison, instead of using our original dataset of 270k users, 26 million ratings, I have decided to use the smaller dataset because it requires less resources and takes a shorter time.

Looking at the results [Appendix 302], all the algo perform better than the the normal predictor. For the surprise library, the normal predictor is an algorithm predicting a random rating based on the distribution of the training set, which is assumed to be normal. T

SVD++ model has the best results but also the longest time. To have a balance between time used and minimal RMSE, I have added a new efficiency calculation which is time*RMSE. The time taken, adds a weight to the RMSE, which looking at the efficiency figure, although the RMSE has a good result, the time taken to run the algorithm is too long.

This is probably because SVD++, the extension of SVD, adds a factor vector for each item and these item factors are used to describe the characteristics of the item. [See Appendix 303 for formula]. It takes into account the user preference so that a better user bias can be obtained.

The models with the next best results are SVD and KNNBaseline. Also, their efficiency is significantly better than SVD++.

The third attempt was to combine the 2 experiments that we have done. This time, a bigger dataset is used – this should improve the RMSE of all the models. And the results reduce the RMSE further which shows a more accurate model but it takes a lot more time. Also notice that with a bigger dataset, the difference between SVD and KNNBaseline is not clear. SVD or SVD++ outperforms any other model in this case.

If we run an RMSE and efficiency comparison, we notice that the RMSE for SVD has the most improvement and the NMF has the least weighted resources consumed. Since ultimately our model aims for the lowest RMSE, the extra time taken to run is negligible and the weighted factor does not increase exponentially, thus, we will look into the SVD model.

Fourth, to look for the peak point of using SVD model. To do this, we will attempt to plot a chart with n-samples against RMSE. This will help to look for the diminishing/increasing return of the model as more data is added. We plot a chart of Datasize VS RSME to get a visual look of the relationship between the 2 [see appendix 305]. From the chart, we can see that there is a diminishing return in adding more data. Due to the intensive computing power needed, we are unable to determine the point to stop.

Next steps:

There is no easy way out of this issue. Like seen above, the way to reduce the RMSE and increase the reliability of the model would be to add more data. This might require cloud computing resources and more money. Another way is to find other libraries which can handle bigger data more efficiently. This will be kept in view for further exploration.

5.2.2. Further research:

While looking at other materials on SVD and collaborative filtering, we tried Tensorflow and manual matrix factorisation. We bump into a lot of problems and upon consultation with Mr Law these needed more exploration, explanation and fine tuning which we were out of time. So we decided to

continue using the surprise library. Details in the experimental log.

5.2.3. Surprise library Part 2:

Data:

Like mentioned previously, the more data we use, the better the results get – RMSE and MAE both reduce as sample size increases. (See Appendix 311). For hyperparameter tuning, we used the sample size of 1,000,000. This is because using the full dataset caused it to crash (See appendix 312).

Setting ***shuffle=True***. One thing is our data is sorted by UserId. Setting ***shuffle=True*** helps to shuffle the UserId. Difference is shown in appendix 319. Notice if ***shuffle=False***, the number of users add back to the original data. This is because our data is sorted by user's id.

Surprise library has their own train_test_split. The testset is output as array while the trainset is output as surprise.dataset. The surprise dataset has an inner id which makes the library function better. This makes data manipulation harder when we want to do our manual testing.

The reason why sample size is important (see appendix 341). As the sample increases, the algo learns more about the latent factors and error is reduced. The final dataset our predictions is ran on has 26,024,289 ratings, 270,896 users and 45,115 movies. A train_test_split is applied - 70% trainset, 30% testset, random_state 42.

Method/preliminary experiments:

Hyperparameter tuning to minimize the error. The model for minimizing error for SVD in the Surprise library is shown in appendix 313. There are 4 keys hyperparameters:
n_epochs = the number of times the SGD procedure is iterated.
lr_all = the learning rate. The learning rate determine how fast the algo moves from 1 epoch to the next.
n_factors = the number of factors in the matrix.
reg_all = regularization factor. Higher means higher variance of predictions. A form of bias-variance tradeoff.

In the surprise library, there is a built-in GridSearchCV which allows different sets of hyperparameters to be run in a cross validation method and it will return the best model based on the parameters with the least error. In addition, it has a hyperparameter “n_jobs” which allows all CPUs to be used in the computation when set to “-1”. The use of multiple CPUs is shown in appendix 314.

There is a difference between cross validation and training the model. In CV, the data is one set, splitted and part of it is used for validation. In this sense, there is no segregation between the test and training set. However, when we are training the model, the train and test set is completely separated.

We ran a few variations (appendix 315) and took the hyperparameters of the best model and put it fit it into the full dataset using the trainset/testset but the values generated are worse than the default setting. (appendix 316-318). One reason is due to the segregation of the train and test set. CV is perfected on itself but changes happen when a completely different set of data is applied. Another reason we can think of is the parameters are not tuned to the smaller dataset used in the tuning phase. A third reason, which I thought of after the presentation, is that we use 1million ratings for gridsearchCV. Our dataset is 270k users and 45k movies. This means that the 1million ratings cannot represent our full dataset because the chance of having repeated user is 27/100 and movies being 45/100. This does not include the permutations of users and movies yet. Given this factor, our 1mil sample data will not be sufficient to represent our 26mil full dataset. As shown in appendix 318, increasing the hyperparameter of “n_factors” from 80 to 120 actually increases the model performance.

MAE vs RMSE. As the variance increases, the difference between MAE and RMSE will increase (appendix 321). So we want to get a balance of it. Notice in Appendix 319 and 322, although the RMSE decreases when reg_all is set to 0.04 from 0.02, the MAE increases. This means the uncertainty of the predictions increases.

As we test other parameters, the higher the numbers, the better the test results. But at some point, it will decrease like in the case of the reg_all. So we will attempt to continue testing with limited permutations since like mentioned in Appendix 312, the PC crashed. Right now, we are trying to find the

point where given our dataset, the point where increasing hyperparameters does not cause a decrease in performance.

After multiple testing and hours of manually changing parameters because of PC RAM limitations issue, we have arrived at a good model. (other details are in the experimental log) Like the gridsearchCV, every parameter will reach a point where it no longer adds value. Our final SVD model has a MAE of 0.5996 and RMSE of 0.7934. Appendix 330 shows a summary of how we derive the hyperparameters.

Recommendations generated: From this model, we will generate the predictions based on the highest estimated rated movie unique to each individual. Example shown on Appendix 331 [fullset in nprecommendedmovies.csv] [individual results in np_prediction_final.csv]

5.2.4. Predictions and accuracy

Prediction test 1: Recommender systems ultimately predict a set of movies that users want to watch. Therefore, I built a function on top of the predicted results to look at the hit rate of the results. Accuracy score is the number of correct predictions in the actual prediction. Since this is based on historical data, we are going to assume that users and movies with ratings are what they have watched. Looking at the results (appendix 342), it is good but not convincing because of the model. Reason being that we are asking the model to pick 10 out of a sample in the testset, not the full movie.

Prediction test 2: Given time constraints, we did another test on the prediction results. This time, we predict the top 10 movies for 1 user, userId 101382, by generating an estimating rating for all movies and then compare it to the top rated movies for the same user. This user has 48 5 stars rating and 8 of our movies are in the list of 48, giving us an accuracy of 80%. Did it on another user, 270893, and it gives 40%. This is a huge variance given only 2 samples.

Prediction test 3: Previously we mentioned that we want to recommend movies that the user has not watched. To do this test, we assume that the training set is what the movies have watched and the testset is what they would watch later. So we apply the model on the movies that the user have not watched [total movie set - movies watched by user in trainset] and compare the result against the testset. Sadly, user 270893 shows a result of 0. (appendix 343), 101382 = 0.4, 70284 = 0.0, 69691 = 0.2. A further test was done on 100 users and the result is 8.6%. This is a meaningful result because the predicted movie list is not in the train set.

5.2.5. Future Work

Like in the 3 prediction tests, there were flaws in each other and with brainstorm, it became better. We can look for a better measure of accuracy. Another work to be done is to introduce a decay to the ratings because of preference changes. Plus, when we were trying out the tensorflow, the RMSE did go down to 0.62 which is better than what we currently have. This is worth a study as well.

5.2.6. Conclusion

We attempt to hire and train the salesman with a logical approach. Firstly, which model is the best and yet does not cost a bomb. Secondly, how do we best train this salesman(model) we hired. More data? Better learning rate? More factors? This is the model that we have with the hyperparameters tuned.

While collaborative model based is effective and we can see its effectiveness, the resource needed to run the program is huge. Furthermore, there is a lot of tuning to be done. In this project, we typically look at reducing the RMSE because it is the metric to gauge if the model is good. However, there are a lot of other metrics to look at like MAE which was used briefly. Given more resources, running gridsearchCV automatically is definitely a plus. Given more time, using Tensorflow seems to be faster but this has yet to be tested. Also, there are other more things to study like prediction accuracy and preference changes by incorporating time decay.

6. Comparison across models

The final movie meta-data features used in content-based filtering contained rich information

	Content-based (euclidean distance,3-NN, weighted average)	Content-based (cosine similarity,3-NN, weighted average)	Memory-based CF - User-based KNN	Memory-based CF - User-based ANN	Memory-based CF - Item-based ANN	Model-based CF SVD
RMSE	1.0623	0.9816	1.0367	0.9666	1.0332	0.7934
MAE	0.8172	0.7442	0.7713	0.7466	0.7623	0.5996
Test set	30,327 predictions (~1000 users in test set)	30,327 predictions (~1000 users in test set)	315 predictions sampled from testset	91 predictions sampled from testset	2285 predictions sampled from testset	The full dataset is used split into 70% trainset, 30% testset. Testset has 7,807,287 ratings, 260,692 users, 34,994 items.
Time	Approximately 4.5 hours, high-ram setting on Colab Pro	Approximately 4.5 hours, high-ram setting on Colab Pro	-	Approximately 0.25hrs for 120 predictions	Approximately 1.5hrs for 4,000 predictions	Many days

In comparing the best model across the three techniques, we find that the Model-based CF SVD out-performed the other models, with an RMSE of 0.7934 and MAE of 0.5996, versus the intermediate-performing Memory-based CF User-based ANN with an RMSE of 0.9666 and MAE of 0.7466, verus the marginally worse-performing Content-based filtering (cosine similarity, 3-NN, weighted average) with an RMSE of 0.9816 and MAE of 0.7442.

For content-based filtering, the method is highly dependent on the movie item features being used. Some features are rather subjective, such as the “overview” column which provides a summary of each movie item. Some overviews go into depth about the movie plot, while other overviews are a single-liner introducing the context to the initial scene. Furthermore, text features are dependent on the type of language being used, which can be subjective in nature, over-use of descriptive flowery language, etc. Another key point is that the historical ratings for each user is critical in determining the predictions output by the recommender. In an instance where the user has only watched and rated movies that do not match his personal taste, the recommender only knows that that user dislikes such films and eliminate similar films from its list of recommendations, but the recommender does not know what movies cater to this user’s preference given there is a lack of history on movies rated highly by that user.

For memory based collaborative filtering, one of the weaknesses is in the distance metric. As discussed throughout this section of the report, there are many concerns that need to be addressed. A way to address this is by using model-based methods, i.e. The SVD method, which instead of looking for similarities with explicit user data like movie ratings, tries to parameterize a set of latent features which might describe and represent each user and movie better.

Another fatal flaw in the nearest neighbour model is the memory and computational time needed. There is heavy use of sampling in this project; unless we have more computational resources, we are unable to actually do the whole test dataset in this project. In contrast, the SVD method is able to do it within reasonable times.

Memory-based models are also easily affected by malicious data, especially for user-based, where a few malicious user-data could easily change the results of the model. Item-based is more resilient due to the fact that it is harder to manipulate movie ratings across multiple users. However, it is also because of this property, a user-based memory model will be able to produce recommendations which are more surprising and diverse; if the user-data aren’t malicious, it will provide new perspective if the data is highly different.

The reason Model based CF SVD has the lowest RMSE is probably due to a combination of movie and user characteristics. While memory based methods are based on either movie or user

similarity and content based is using attributes inputted from the programmer, SVD figures out the latent factors on its own. Thus, there could be some hidden features that are not discovered in content based. For SVD, the optimal model uses 120 factors but for content based, features are manually discerned and input.

In terms of future exploration beyond the project, a hybrid approach that combines the use of content-based and collaborative methods can bring out the strength of both methods is the likely winner. Different implementations include combining recommendations from both methods, using collaborative data as a feature in content-based recommenders and using content-based predictor to complete collaborative data [106].

7. Deployment

Output and usage of output

Output of a list of recommendations in different categories (e.g. genre of movie). Recommendations output to be changed periodically with a refreshed list, and the model to be updated periodically according to users' latest viewing history.

Individual contributions

21A471K Shee Jing Le	Section 1. Introduction Section 5.2.2 Collaborative filtering model-based Section 6. Comparison across models
21A466A Ong Joo Kiat, Kenneth	Section 2.1.1 User ratings data Section 4. Evaluating recommender systems Section 5.2.1 Collaborative filtering memory-based Section 6. Comparison across models
21A530M Pamela Sin Hui	Section 2. Dataset Section 4 Content-based filtering Section 6 Comparison across models

Discussion of the topic, sourcing of data, combining data and matrix discussion was done together as a team.

References:

[1] F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. ACM Transactions on Interactive Intelligent Systems (TiiS)

[2] Denis Kotkov, Shuaiqiang Wang, Jari Veijalainen. 2016. A Survey of Serendipity in Recommender Systems. Knowledge-Based Systems

[101] Building a Recommender System - KDnuggets. (2021). Retrieved 28 August 2021, from https://www.kdnuggets.com/2019/04/building-recommender-system.html

[102] (2021). Retrieved 24 August 2021, from https://sites.cs.ucsb.edu/~tyang/class/293S17/slides/Topic8RecommendSimple.pdf

[103] Introduction to TWO approaches of Content-based Recommendation System. (2021). Retrieved 24 August 2021, from https://towardsdatascience.com/introduction-to-two-approaches-of-content-based-recommendation-system-fc797460c18c

[104] Introduction To Recommender Systems- 1: Content-Based Filtering And Collaborative Filtering. (2021). Retrieved 24 August 2021, from https://towardsdatascience.com/introduction-to-recommender-systems-1-971bd274f421

[105] (2021). Retrieved 24 August 2021, from https://www.datacamp.com/community/tutorials/recommender-systems-pytho

[107] sklearn.feature_extraction.text.CountVectorizer — scikit-learn 0.24.2 documentation. (2021). Retrieved 24 August 2021, from https://scikit-learn.org/stable/m

odules/generated/sklearn.feature_extraction.text.CountVectorizer.html

[108] Stop Words in NLP. (2021). Retrieved 24 August 2021, from https://medium.com/@saitejaponugoti/stop-words-in-nlp-5b248dadad47

[**109]** sklearn.feature_extraction.text.TfidfTransformer — scikit-learn 0.24.2 documentation. (2021). Retrieved 24 August 2021, from https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfTransformer.html#sklearn.feature_extraction.text.TfidfTransformer

[**110]** sklearn.metrics.pairwise.euclidean_distances — scikit-learn 0.24.2 documentation. (2021). Retrieved 24 August 2021, from

https://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.euclidean_distances.html

[201] Alexandr Andoni, Piotr Indyk and Ilya Razenshteyn, 2018, Approximate Nearest Neighbor Search in High Dimensions, ArXiv.

[202] Christian Desrosiers and George Karypis, 2011, A Comprehensive Survey of Neighborhood-based Recommendation Methods, Recomm*ender Systems Handbook.*

[203] Noi Sian Koh, Nan Hu, Eric K. Clemons, 2010, Do online reviews reflect a product's true perceived quality? An investigation of online movie reviews across cultures, *Research Collection School Of* Information Systems.

[301] "Recommender System—singular value decomposition (SVD) & truncated SVD", *Medium*, 2021. [Online]. Available: https://towardsdatascience.com/recommender-system-singular-value-decomposition-svd-truncated-svd-97096338f361. [Accessed: 24- Aug- 2021].

[302] Z. Xian, 2021. [Online]. Available: https://www.hindawi.com/journals/mpe/2017/1975719/. [Accessed: 24- Aug- 2021].

[303]"Welcome to Surprise' documentation! — Surprise 1 documentation", *Surprise.readthedocs.io*, 2021. [Online]. Available: https://surprise.readthedocs.io/en/stable/index.html. [Accessed: 24- Aug- 2021].

[304]"Matrix Factorization-based algorithms — Surprise 1 documentation", *Surprise.readthedocs.io*, 2021. [Online]. Available: https://surprise.readthedocs.io/en/stable/matrix_factorization.html#unbiased-note. [Accessed: 24- Aug- 2021].

[305] "Matrix Factorization-based algorithms — Surprise 1 documentation", *Surprise.readthedocs.io*, 2021. [Online]. Available: https://surprise.readthedocs.io/en/stable/matrix_factorization.html#unbiased-note. [Accessed: 24- Aug- 2021].

[306] D. Kumar, "Singular Value Decomposition (SVD) & Its Application In Recommender System", *Analytics India Magazine*, 2021. [Online]. Available: https://analyticsindiamag.com/singular-value-decomposition-svd-application-recommender-system/. [Accessed: 24- Aug- 2021].

Appendix:

Appendix 101 - Example of transforming a raw feature

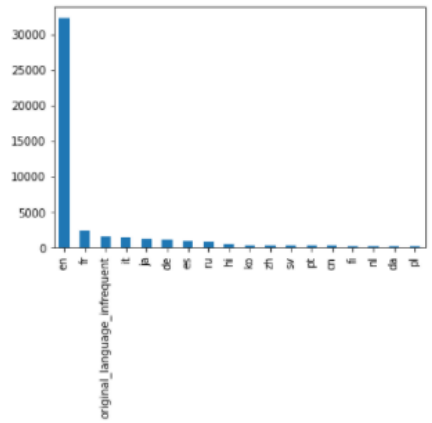
```
# For original_language that have less than 200 rows for that category, we change cells to a single string "original_language_infrequent"
movies_df.loc[movies_df['original_language'].isin(temp_list) ,'original_language'] = 'original_language_infrequent'
pd.concat([movies_df.original_language.value_counts().to_frame(), movies_df.original_language.value_counts(normalize=True).to_frame()], axis=1)
```

	original_language	original_language
en	32233	0.710103
fr	2434	0.053622
original_language_infrequent	1599	0.035226
it	1529	0.033684
ja	1344	0.029609
...	...	...
cn	313	0.006895
fi	294	0.006477
nl	248	0.005464
da	223	0.004913
pl	219	0.004825

18 rows × 2 columns

```
movies_df['original_language'].value_counts().plot(kind='bar')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4c6aea2290>
```



Appendix 102 - Non-text features

```
#1. Creating non_text_features_sparse_matrix
import scipy
non_text_features_sparse_matrix = scipy.sparse.csr_matrix(movies_df2[['belongs_to_collection', 'homepage', 'same_title', 'runtime_scaled']].copy())
non_text_features_sparse_matrix
```

```
<45403x4 sparse matrix of type '<class 'numpy.float64'>'
        with 91677 stored elements in Compressed Sparse Row format>
```

```
type(non_text_features_sparse_matrix)
```

```
scipy.sparse.csr.csr_matrix
```

```
non_text_features_sparse_matrix.shape
```

```
(45403, 4)
```

Appendix 103 - Objective data final “soup” feature

```
#Creation of 'soup' text, one of the final movie features, which is created by:
#1. Top 8 keywords
#2. Top 3 cast
#3. Director
#4. Top 3 genres
#5. Top 3 production companies
#6. Top 3 production countries
#7. original language
movies_df2[['title', 'cast', 'director', 'keywords', 'genres', 'production_companies_cleaned', 'production_countries_cleaned', 'original_language', 'soup']].head(10)
```

	title	cast	director	keywords	genres	production_companies_cleaned	production_countries_cleaned	original_language	soup
0	Toy Story	[tomhanks, timallen, donrickles]	johnlasseter	[jealousy, toy, boy, friendship, friends, riva...	[animation, comedy, family]	[pixaranimationstudios]	[unitedstatesofamerica]	[en]	jealousy toy boy friendship friends rivalry bo...
1	Jumanji	[robinwilliams, jonathanhyde, kirstendunst]	joejohnston	[boardgame, disappearance, basedonchildren'sbo...	[adventure, fantasy, family]	[tristarpictures, teitlerfilm, interscopecommu...	[unitedstatesofamerica]	[en]	boardgame disappearance basedonchildren'sbook ...
2	Grumpier Old Men	[waltermatthau, jacklemmon, ann-margret]	howarddeutch	[fishing, bestfriend, duringcreditsstinger, ol...	[romance, comedy]	[warnerbros., lancastergate]	[unitedstatesofamerica]	[en]	fishing bestfriend duringcreditsstinger oldmen...
3	Waiting to Exhale	[whitneyhouston, angelabassett, lorettadevine]	forestwhitaker	[basedonnovel, interracialrelationship, single...	[comedy, drama, romance]	[twentiethcenturyfoxfilmcorporation]	[unitedstatesofamerica]	[en]	basedonnovel interracialrelationship singlemot...
4	Father of the Bride Part II	[stevemartin, dianekeaton, martinshort]	charlesshyer	[baby, midlifecrisis, confidence, aging, daugh...	[comedy]	[sandollarproductions, touchstonepictures]	[unitedstatesofamerica]	[en]	baby midlifecrisis confidence aging daughter m...
5	Heat	[alpacino, robertdeniro, valkilmer]	michaelmann	[robbery, detective, bank, obsession, chase, s...	[action, crime, drama]	[regencyenterprises, forwardpass, warnerbros.]	[unitedstatesofamerica]	[en]	robbery detective bank obsession chase shootin...
6	Sabrina	[harrisonford, juliaormond, gregkinnear]	sydneypollack	[paris, brotherbrotherrelationship, chauffeur...	[comedy, romance]	[paramountpictures, scottrudinproductions, mir...	[germany, unitedstatesofamerica]	[en]	paris brotherbrotherrelationship chauffeur lon...
7	Tom and Huck	[jonathantaylorthomas, bradrenfro, rachaelleig...	peterhewitt	[]	[action, adventure, drama]	[waltdisneypictures]	[unitedstatesofamerica]	[en]	jonathantaylorthomas bradrenfro rachaelleighc...
8	Sudden Death	[jean-claudevandamme, powersboothe, dorianhare...	peterhyams	[terrorist, hostage, explosive, vicepresident]	[action, adventure, thriller]	[universalpictures, imperialentertainment, sig...	[unitedstatesofamerica]	[en]	terrorist hostage explosive vicepresident jean...
9	GoldenEye	[piercebrosnan, seanbean, izabellascorupco]	martincampbell	[cuba, falselyaccused, secretidentity, compute...	[adventure, action, thriller]	[unitedartists, eonproductions]	[unitedkingdom, unitedstatesofamerica]	[en]	cuba falselyaccused secretidentity computervir...

Appendix 104 - Objective features (Count vectoriser)

```
#2. Create count_matrix from soup
from sklearn.feature_extraction.text import CountVectorizer

count = CountVectorizer(stop_words='english')
count_matrix = count.fit_transform(movies_df2['soup'])
```

```
type(count_matrix)
```

```
scipy.sparse.csr.csr_matrix
```

```
count_matrix.shape
```

```
(45403, 111666)
```

Appendix 105 - Subjective features (TF-IDF vectoriser)

```
#3. Create tfidf_matrix from overview_cleaned
```

```
movies_df2['overview_cleaned'] = movies_df2.overview_cleaned.astype(str)
```

```
movies_df2['overview_cleaned']
```

```
0        led woodi andi toy live happili room andi birt...
1        sibl judi peter discov enchant board game open...
2        famili wed reignit ancient feud neighbor fish ...
3        cheat mistreat step women hold breath wait elu...
4        georg bank recov daughter wed receiv news preg...
                              ...
45398                                rise fall man woman
45399    artist struggl finish work storylin cult play ...
45400    one hit goe wrong profession assassin end suit...
45401    small town live two brother one minist one hun...
45402    year decriminalis homosexu uk director daisi a...
Name: overview_cleaned, Length: 45403, dtype: object
```

```
tfidf= TfidfVectorizer()
tfidf_matrix = tfidf.fit_transform(movies_df2['overview_cleaned'])
```

```
type(tfidf_matrix)
```

```
scipy.sparse.csr.csr_matrix
```

```
tfidf_matrix.shape
```

```
(45403, 55118)
```

Appendix 106 - Deriving similarity matrix in batches of items

```
# Compute the Cosine Similarity matrix based on the count_matrix
from sklearn.metrics.pairwise import cosine_similarity
#cosine_sim = cosine_similarity(features_sparse_matrix, features_sparse_matrix)
```

```
#https://stackoverflow.com/questions/40900608/cosine-similarity-on-large-sparse-matrix-with-numpy
# Prevent colab from crashing when calculating cosine similarity

def cosine_sim_function(m1, m2, batch_size=10):
    assert m1.shape[1] == m2.shape[1]
    ret = np.ndarray((m1.shape[0], m2.shape[0]))
    for row_i in range(0, int(m1.shape[0] / batch_size) + 1):
        start = row_i * batch_size
        end = min([(row_i + 1) * batch_size, m1.shape[0]])
        if end <= start:
            break # edge cases
        rows = m1[start: end]
        sim = cosine_similarity(rows, m2)
        ret[start: end] = sim
    return ret

cosine_sim =cosine_sim_function(features_sparse_matrix, features_sparse_matrix)
```

Appendix 107 - Simple and weighted average against cosine similarity

```
for each_test_userId in ratings2_test_userId_list:

 # For the userId, find the ratings data that falls in the train set and test set respectively
 user_1_train = ratings2_train.loc[ratings2_train.userId == each_test_userId,]
 user_1_test = ratings2_test.loc[ratings2_test.userId == each_test_userId,]

 # From the ratings data train set for that userId, get the list of movie indexes (a user only rates a movie once, so there is no repeat movie index)
 user_1_train_indexes_list = list(user_1_train['index'])

 # For that userId, the goal is to predict ratings for each movie in the test set
 for index, row in user_1_test.iterrows():
   #For that userId, iterate through each row (movie) of the test set
   #Create an empty list to store the simillilarity scores of each movie in the train set for that userId, to that movie in question
   sim_scores_train = []

   #index for that movie of the test set
   idx = int(row['index'])
   #This is the similarity scores of that movie of the test set to the global movies
   sim_scores = list(enumerate(cosine_sim[idx]))

   #We only store the tuples that indicate the similarity scores of each movie in the train set for that userId, to that movie in question
   for each_tuple in sim_scores:
     if each_tuple[0] in user_1_train_indexes_list:
       sim_scores_train.append(each_tuple)

   #Sort similarity scores from highest to lowest similarity
   sim_scores_train = sorted(sim_scores_train, key=lambda x: x[1], reverse=True)

   #Defining variables
   simple_average = 0
   weighted_average = 0
   the_index = 0
   the_similarity = 0
   the_rating = 0
   the_similarities = []
   the_ratings = []

   print('Tuples', sim_scores_train[:3])
   #For each of the top three movies in the train set for that userID to that movie in question in the test set, store the similarity in a list and store the rating in a list
   for each_tuple in sim_scores_train[:3]:
     the_index = each_tuple[0]
     the_similarity = float(each_tuple[1])
     the_rating = float(user_1_train.loc[user_1_train['index'] == the_index, 'rating'])
     simple_average += the_rating

     the_similarities.append(the_similarity)
     the_ratings.append(the_rating)

   #Sum of similarities of the top three movies in the train set for that userID to that movie in question in the test set
   the_similarities_sum = sum(the_similarities)
   try:
     simple_average /= len(the_similarities)
     for i in range(len(the_similarities)):
       weighted_average += the_similarities[i] * the_ratings[i] / the_similarities_sum

   except:
     simple_average = np.nan #"Not obtained"
     weighted_average = np.nan #"Not obtained"

   print('the_similarities',the_similarities)
   print('the_ratings',the_ratings)
   print('simple_average',simple_average)
   print('weighted_average', weighted_average)

   ratings2_test.loc[(ratings2_test.userId == each_test_userId) & (ratings2_test['index'] == idx) ,'cosine_sim_simple_average'] = simple_average
   ratings2_test.loc[(ratings2_test.userId == each_test_userId) & (ratings2_test['index'] == idx),'cosine_sim_weighted_average'] = weighted_average
```

Appendix 108 - Results of content based filtering (cosine similarity)

```
[172] # See the predicted ratings
      ratings_cosine_sim = ratings2_test.loc[ratings2_test.cosine_sim_simple_average.notnull()].copy()
      ratings_cosine_sim
```

	userId	movieId	rating	timestamp	index	cosine_sim_simple_average	cosine_sim_weighted_average
21100154	660	1348	3.5	1119627904	1297	3.833333	3.978188
13030836	775	112	4.0	1108150261	110	3.833333	3.829531
8382153	1025	1196	3.0	1433113508	1151	2.333333	2.474749
23059686	390	6197	4.0	1048394998	6048	3.666667	3.673065
22407078	229	4009	3.0	1037824011	3879	3.000000	3.064496
...	...	...	...	...	...	...	...
10664422	602	41997	3.0	1167189834	10624	4.166667	4.166240
2117969	904	3578	4.0	1029086942	3451	3.666667	3.669167
7758160	321	232	4.5	1182059630	228	3.333333	3.311428
2954856	173	2571	5.0	1443090717	2454	4.333333	4.203279
14141116	728	2247	3.0	976226395	2132	2.666667	2.669713

30327 rows × 7 columns

```
[173] from sklearn.metrics import mean_squared_error
      print("RMSE using simple_average (cosine similarity) 3 nearest neighbors:", np.sqrt(mean_squared_error(ratings_cosine_sim.rating, ratings_cosine_sim.cosine_sim_simple_average)))
      print("RMSE using weighted_average (cosine similarity) 3 nearest neighbors:", np.sqrt(mean_squared_error(ratings_cosine_sim.rating, ratings_cosine_sim.cosine_sim_weighted_average)))

RMSE using simple_average (cosine similarity) 3 nearest neighbors: 0.9856360774602401
RMSE using weighted_average (cosine similarity) 3 nearest neighbors: 0.981609078683196
```

```
[ ] from sklearn.metrics import mean_absolute_error
    print("MAE using simple_average (cosine similarity) 3 nearest neighbors:", mean_absolute_error(ratings_cosine_sim.rating, ratings_cosine_sim.cosine_sim_simple_average))
    print("MAE using weighted_average (cosine similarity) 3 nearest neighbors:", mean_absolute_error(ratings_cosine_sim.rating, ratings_cosine_sim.cosine_sim_weighted_average))

MAE using simple_average (cosine similarity) 3 nearest neighbors: 0.7471010430749279
MAE using weighted_average (cosine similarity) 3 nearest neighbors: 0.7441933637051575
```

Appendix 109 - Results of content based filtering (euclidean distance)

```
[165] # See the predicted ratings
      ratings_euclidean_dist = ratings2_test.loc[ratings2_test.euclidean_dist_simple_average.notnull()].copy()
      ratings_euclidean_dist
```

	userId	movieId	rating	timestamp	index	euclidean_dist_simple_average	euclidean_dist_weighted_average
21100154	660	1348	3.5	1119627904	1297	3.833333	3.863006
13030836	775	112	4.0	1108150261	110	3.166667	3.170523
8382153	1025	1196	3.0	1433113508	1151	2.333333	2.392366
23059686	390	6197	4.0	1048394998	6048	1.500000	1.504508
22407078	229	4009	3.0	1037824011	3879	2.666667	2.665237
...	...	...	...	...	...	...	...
10664422	602	41997	3.0	1167189834	10624	4.166667	4.153496
2117969	904	3578	4.0	1029086942	3451	4.000000	3.999439
7758160	321	232	4.5	1182059630	228	2.500000	2.499000
2954856	173	2571	5.0	1443090717	2454	4.166667	4.150149
14141116	728	2247	3.0	976226395	2132	2.666667	2.666783

30327 rows × 7 columns

```
[168] from sklearn.metrics import mean_squared_error
      print("RMSE using simple_average (euclidean distance) 3 nearest neighbors:", np.sqrt(mean_squared_error(ratings_euclidean_dist.rating, ratings_euclidean_dist.euclidean_dist_simple_average)))
      print("RMSE using weighted_average (euclidean distance) 3 nearest neighbors:", np.sqrt(mean_squared_error(ratings_euclidean_dist.rating, ratings_euclidean_dist.euclidean_dist_weighted_average)))

      RMSE using simple_average (euclidean distance) 3 nearest neighbors: 1.0642148829033191
      RMSE using weighted_average (euclidean distance) 3 nearest neighbors: 1.0623319934360311
```

```
[167] from sklearn.metrics import mean_absolute_error
      print("MAE using simple_average (euclidean distance) 3 nearest neighbors:", mean_absolute_error(ratings_euclidean_dist.rating, ratings_euclidean_dist.euclidean_dist_simple_average))
      print("MAE using weighted_average (euclidean distance) 3 nearest neighbors:", mean_absolute_error(ratings_euclidean_dist.rating, ratings_euclidean_dist.euclidean_dist_weighted_average))

      MAE using simple_average (euclidean distance) 3 nearest neighbors: 0.8187698530462404
      MAE using weighted_average (euclidean distance) 3 nearest neighbors: 0.8172078491901094
```

Appendix 110 - Features of commonly recommended movies

```
# When viewing recommended movies for users, we notice that some movies tend to be commonly recommended.
# Viewing the features of these commonly recommended movies, we notice that the "soup" column tends to be short with common words with other movies like "unitedstatesofamericaen".
# The lack of other words and the appearance of common words is perhaps what leads to a resultant close similarity with other movies.
commonly_recommended = ['The Drunk','Superpower', 'Το γύλο','Saved by the Bell: Wedding in Las Vegas',"The First Annual 'On Cinema' Oscar Special","The Fourth Annual 'On Cinema' Oscar Special", 'Made For Each Other', 'Made For Each Other']
movies_df2.loc[movies_df2.title.isin(commonly_recommended),['title','belongs_to_collection', 'homepage', 'same_title', "runtime_scaled",'soup', 'overview_cleaned']]
```

	title	belongs_to_collection	homepage	same_title	runtime_scaled	soup	overview_cleaned
22963	Superpower	0.0	1.0	1.0	0.652317	documentarysuperpowerproductions unitedstat...	superpow illustr unit state leverag posit ensu...
23199	Made For Each Other	0.0	0.0	1.0	0.164675	comedy romance unitedstatesofamericaen	pair creat lover stranger take marriag impos
30343	The Drunk	0.0	1.0	1.0	-0.091980	unitedstatesofamericaen	hard drink grandson legendari labor leader get...
34621	Το γύλο	0.0	0.0	1.0	0.139009	drama original_language_infrequent	greek movi famili main focus troubl
40066	Saved by the Bell: Wedding in Las Vegas	0.0	0.0	1.0	-0.091980	unitedstatesofamericaen	conclus long run seri final happen kelli zack ...
40365	The First Annual 'On Cinema' Oscar Special	0.0	0.0	1.0	0.421329	comedy unitedstatesofamericaen	first annual cinema oscar special
42533	The Fourth Annual 'On Cinema' Oscar Special	0.0	0.0	1.0	2.348234	comedy unitedstatesofamericaen	tim showcas band dekkar find closur tom cruis ...

Appendix 301 - Surprise dataset size results

```
svd = SVD()
knn = KNNBasic()
svdpp = SVDpp()
epoch = 10000
summary = []

while epoch <= 50000:
    ratingsmini = ratings.sample(n=epoch, replace=False, random_state=42)
    rawmini = Dataset.load_from_df(ratingsmini[['userId','movieId','rating']],reader)
    trainset, testset = train_test_split(rawmini, test_size=.3, random_state=42)
    out = cross_validate(svd, rawmini, measures=['RMSE', 'MAE'], cv=5, verbose=True)
    mean_rmse = '{:.3f}'.format(np.mean(out['test_rmse']))
    summary.append(["n= ", epoch, 'mean_rmse = ', mean_rmse])
    print("n= ", epoch, 'mean_rmse = ', mean_rmse)
    epoch += 10000

print(summary)
```

```
[['n= ', 10000, 'mean_rmse = ', '1.035'], ['n= ', 20000, 'mean_rmse = ', '1.020'], ['n= ', 30000, 'mean_rmse = ', '1.012'], ['n
= ', 40000, 'mean_rmse = ', '1.002'], ['n= ', 50000, 'mean_rmse = ', '0.997']]
```

Appendix 302 - Surprise model results 100k dataset

```
dataset = 'ml-100k'
data = Dataset.load_builtin(dataset)
```

```
for klass in classes:
    start = time.time()
    out = cross_validate(klass(), data, ['rmse', 'mae'], kf)
    cv_time = str(datetime.timedelta(seconds=int(time.time() - start)))
    link = LINK[klass.__name__]
    mean_rmse = '{:.3f}'.format(np.mean(out['test_rmse']))
    mean_mae = '{:.3f}'.format(np.mean(out['test_mae']))
    efficient  = str(np.mean(out['test_rmse'])*int(time.time() - start))

    new_line = [link, mean_rmse, mean_mae, cv_time, efficient]
    #print(tabulate([new_line], tablefmt="pipe"))  # print current algo perf
    table.append(new_line)

header = [LINK[dataset],
          'RMSE',
          'MAE',
          'Time',
          'Efficiency'
          ]
print(tabulate(table, header, tablefmt="pipe"))
```

ml-100k	RMSE	MAE	Time	Efficiency
SVD	0.936	0.738	0:00:13	12.1689
SVDpp	0.919	0.722	0:07:41	423.801
NMF	0.964	0.758	0:00:15	14.4581
SlopeOne	0.944	0.742	0:00:09	8.49888
KNNBasic	0.979	0.773	0:00:10	9.78931
KNNWithMeans	0.95	0.749	0:00:11	10.4518
KNNBaseline	0.93	0.733	0:00:13	12.0893
CoClustering	0.966	0.757	0:00:06	5.79391
BaselineOnly	0.944	0.748	0:00:01	0.943637
NormalPredictor	1.522	1.222	0:00:01	1.52214

Appendix 303 – Surprise formula SVD vs SVD++

SVD

$$\tilde{r}_{ui} = \mu + b_u + b_i + q_i^T \cdot p_u$$

SVD++

$$\tilde{r}_{ui} = \mu + b_u + b_i + q_i^T \cdot \left(p_u + |R(u)|^{-1/2} \sum_{j \in R(u)} y_j \right)$$

Appendix 304 – Surprise model results with 1m dataset

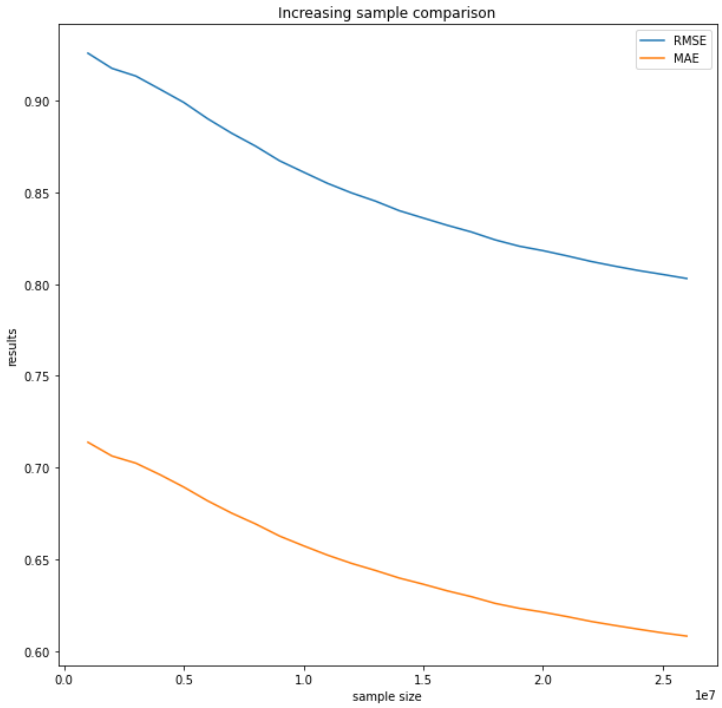
```
dataset = 'ml-1m'
data = Dataset.load_builtin(dataset)
```

ml-1m	RMSE	MAE	Time	Efficiency
SVD	0.874	0.686	0:02:13	116.204
SVDpp	0.862	0.672	2:30:04	7761.04
NMF	0.916	0.724	0:02:28	135.631
SlopeOne	0.907	0.714	0:03:07	169.517
KNNBasic	0.923	0.727	0:07:21	406.965
KNNWithMeans	0.929	0.738	0:07:29	417.188
KNNBaseline	0.895	0.706	0:07:54	424.176
CoClustering	0.916	0.718	0:01:10	64.0868
BaselineOnly	0.909	0.719	0:00:15	13.6293
NormalPredictor	1.506	1.207	0:00:16	24.0972

Appendix 305 – Comparison of 100k and 1m

	RMSE			Efficiency		
	100k	1m	Improvement	100k	1m	Weighted time factor
SVD	0.936	0.874	7.09%	12.1689	116.204	9.55
SVDpp	0.919	0.862	6.61%	23.801	7761.04	326.08
NMF	0.964	0.916	5.24%	14.4581	135.631	9.38
SlopeOne	0.944	0.907	4.08%	8.49888	169.517	19.95
KNNBasic	0.979	0.923	6.07%	9.78931	406.965	41.57
KNNWithMeans	0.95	0.929	2.26%	10.4518	417.188	39.92
KNNBaseline	0.93	0.895	3.91%	12.0893	424.176	35.09
CoClustering	0.966	0.916	5.46%	5.79391	64.0868	11.06
BaselineOnly	0.944	0.909	3.85%	0.943637	13.6293	14.44
NormalPredictor	1.522	1.506	1.06%	1.52214	24.0972	15.83

Appendix 311 – Surprise increasing sample



Appendix 312 – Gridsearch crash (*raw* is the full dataset of 27,000,000 ratings)

```
from surprise.model_selection import GridSearchCV

grid = {'n_epochs': [5, 10, 15, 20, 25, 30],
        'lr_all': [.001, .0025, .005, .0075, .01],
       'n_factors': [80, 90, 100, 110, 120]}

gs = GridSearchCV(SVD, grid, measures=['rmse', 'mae'], cv=5, n_jobs= -1)
gs.fit(raw)

print(gs.best_score['mae'])
print(gs.best_score['rmse'])
print(gs.best_params['mae'])
print(gs.best_params['rmse'])
```

```
C:\Users\sjlal\anaconda3\lib\site-packages\joblib\externals\loky\process_executor.py:688: UserWarning: A worker stopped while s
ome jobs were given to the executor. This can be caused by a too short worker timeout or by a memory leak.
  warnings.warn(
```

Appendix 313 – Surprise library SVD hyperparameter tuning

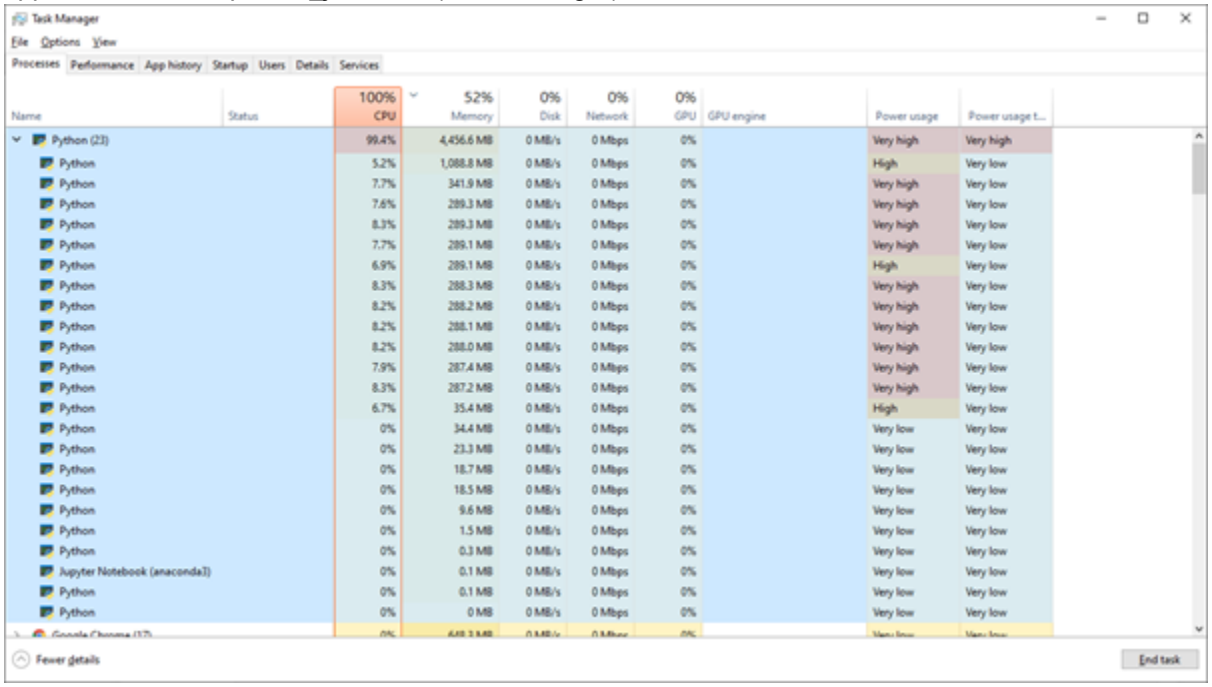
To estimate all the unknown, we minimize the following regularized squared error:

$$\sum_{r_{ui} \in R_{train}} (r_{ui} - \hat{r}_{ui})^2 + \lambda \left(b_i^2 + b_u^2 + ||q_i||^2 + ||p_u||^2 \right)$$

The minimization is performed by a very straightforward stochastic gradient descent:

$$\begin{aligned} b_u &\leftarrow b_u + \gamma(e_{ui} - \lambda b_u) \\ b_i &\leftarrow b_i + \gamma(e_{ui} - \lambda b_i) \\ p_u &\leftarrow p_u + \gamma(e_{ui} \cdot q_i - \lambda p_u) \\ q_i &\leftarrow q_i + \gamma(e_{ui} \cdot p_u - \lambda q_i) \end{aligned}$$

Appendix 314 – surprise n_jobs = -1 (Task Manager)



Appendix 315 – GridSearchCV results sample

```
sample = 1000000

grid = {'n_epochs': [5, 10, 15, 20, 25, 30],
        'lr_all': [.001, .0025, .005, .0075, .01],
       'n_factors': [80, 90, 100, 110, 120]}

gs = GridSearchCV(SVD, grid, measures=['rmse', 'mae'], cv=5, n_jobs= -1)
ratingsmini = ratings.sample(n=sample, replace=False, random_state=42)
reader = Reader()
rawmini = Dataset.load_from_df(ratingsmini[['userId','movieId','rating']],reader)
gs.fit(rawmini)

print(gs.best_score['mae'])
print(gs.best_score['rmse'])
print(gs.best_params['mae'])
print(gs.best_params['rmse'])
```

```
0.7115954738901158
0.9229883921706896
{'n_epochs': 30, 'lr_all': 0.0025, 'n_factors': 80}
{'n_epochs': 30, 'lr_all': 0.0025, 'n_factors': 80}
```

Appendix 319 – Shuffle=True

Train and test set

```
trainset, testset = train_test_split(raw, test_size=.3, random_state=42, shuffle=True)
```

```
print("Number of ratings = ", ratings['userId'].count())
print("Number of Users = ", ratings['userId'].nunique())
print("Number of Movies = ", ratings['movieId'].nunique())
```

```
Number of ratings =  26024289
Number of Users =  270896
Number of Movies =  45115
```

```
print("trainset users= ", trainset.n_users)
print("trainset items= ", trainset.n_items)
print("trainset ratings= ", trainset.n_ratings)

nptestset = np.array(testset)
nptestset[:,1]
testsetn_items=len(np.unique(nptestset[:,1]))
testsetn_users=len(np.unique(nptestset[:,0]))
testsetn_ratings=len(nptestset)

print("testset users= ", testsetn_users)
print("testset items= ", testsetn_items)
print("testset ratings= ", testsetn_ratings)
```

```
trainset users=  268903
trainset items=  42277
trainset ratings=  18217002
testset users=  260692
testset items=  34994
testset ratings=  7807287
```

Appendix 319 – Shuffle=False

Train and test set

```
trainset, testset = train_test_split(raw, test_size=.3, random_state=42, shuffle=False)
```

```
print("Number of ratings = ", ratings['userId'].count())
print("Number of Users = ", ratings['userId'].nunique())
print("Number of Movies = ", ratings['movieId'].nunique())
```

```
Number of ratings =  26024289
Number of Users =  270896
Number of Movies =  45115
```

```
print("trainset users= ", trainset.n_users)
print("trainset items= ", trainset.n_items)
print("trainset ratings= ", trainset.n_ratings)

nptestset = np.array(testset)
nptestset[:,1]
testsetn_items=len(np.unique(nptestset[:,1]))
testsetn_users=len(np.unique(nptestset[:,0]))
testsetn_ratings=len(nptestset)

print("testset users= ", testsetn_users)
print("testset items= ", testsetn_items)
print("testset ratings= ", testsetn_ratings)
```

```
trainset users=  188980
trainset items=  42185
trainset ratings=  18217002
testset users=  81917
testset items=  35544
testset ratings=  7807287
```

Appendix 322 - MAE vs RMSE

CASE 1: Evenly distributed errors

ID	Error	|Error|	Error^2
1	2	2	4
2	2	2	4
3	2	2	4
4	2	2	4
5	2	2	4
6	2	2	4
7	2	2	4
8	2	2	4
9	2	2	4
10	2	2	4

MAE	RMSE
2.000	2.000

CASE 2: Small variance in errors

ID	Error	|Error|	Error^2
1	1	1	1
2	1	1	1
3	1	1	1
4	1	1	1
5	1	1	1
6	3	3	9
7	3	3	9
8	3	3	9
9	3	3	9
10	3	3	9

MAE	RMSE
2.000	2.236

CASE 3: Large error outlier

ID	Error	|Error|	Error^2
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0
6	0	0	0
7	0	0	0
8	0	0	0
9	0	0	0
10	20	20	400

MAE	RMSE
2.000	6.325

MAE and RMSE for cases of increasing error variance

Appendix 330 – SVD manual testing summary and conclusion

Sample size full (~25m)							
	Epoch	LR	Factors	Reg	MAE	RMSE	
Default	20	0.005	100	0.02	0.6083	0.8086	Comments
Using HP of Gridsearch with sample	30	0.0025	80	0.02	0.6134	0.8079	Worst than default
Increase factors	30	0.0025	120	0.02	0.6131	0.8074	Improvement but marginal
Increase LR	30	0.005	120	0.02	0.6062	0.8029	Improvement and significant
Increase Reg	30	0.005	120	0.04	0.6080	0.8018	Improvement RMSE but not MAE
Decrease Reg	30	0.005	120	0.03	0.6026	0.7966	Improvement RMSE and MAE from reg = 0.02
Increase factors	30	0.005	140	0.03	0.6025	0.7964	Improvement but marginal
Increase factors	30	0.005	160	0.03	0.6024	0.7962	Improvement but marginal
Factors = 120, Increase Epoch	40	0.005	120	0.03	0.5997	0.7938	Improvement and significant
Factors = 120, Increase Epoch	50	0.005	120	0.03	0.5996	0.7940	Marginal Improvement MAE but not RSME
Factors = 120, Increase LR	40	0.006	120	0.03	0.6006	0.7953	No Improvement
3 HP fixed, testing factors	40	0.005	160	0.03	0.5996	0.7934	Marginal RMSE Improvement
3 HP fixed, testing factors	40	0.005	200	0.03	0.5998	0.7934	No Improvement
3 HP fixed, testing factors	40	0.005	180	0.03	0.5998	0.7935	No Improvement

Appendix 331 – predictions

```
In [57]: recommendedmovies

Out[57]: [[101382, [858, 1221, 527, 1228, 356, 1213, 1393, 953, 150, 508]],
          [70284, [3147, 2858, 1246, 4995, 3408, 1357, 3844, 1097, 1641, 1784]],
          [87872, [2628, 912, 902, 2303, 355]],
          [69691, [6016, 3147, 3949, 110, 5995, 4011, 4993, 1784, 293, 2997]],
          [95304, [908, 912, 926, 1254, 1262, 1203, 266, 457, 1204, 1248]],
          [123494, [318, 356, 912, 151707, 1196, 750, 2019, 296, 1252, 1197]],
          [252050, [3107, 1805, 849, 1093, 3210, 273]],
          [63441, [1527, 2324, 2858, 1704, 527, 1680, 32, 1198, 529, 1748]],
          [164067, [7, 539, 276, 39, 380, 3, 296, 420, 367, 410]],
          [191886, [3996, 1784, 3793]],
          [79039, [318, 2571, 3578, 4878, 4011]],
          [95135, [527, 1193, 1227, 356, 7147, 3198, 3252, 6874, 1209, 7438]],
          [41562, [296, 1148, 745, 778, 1199, 1206, 1136, 1288, 318, 1358]],
          [108770, [858, 527, 1207, 745, 1204, 1148, 1234, 608, 913, 1259]],
          [258021, [858, 1228, 2288, 1997, 2858, 1387, 3471, 2318, 1222, 3168]],
          [229359, [50, 858, 3147, 4226, 4011, 3578, 5952, 593, 293, 3275]],
          [75291, [58559, 134130, 79132, 50, 5989, 84152, 8368, 72998, 106642, 3256]],
          [126956,
           [79091, 79132, 76093, 77561, 58559, 109487, 91529, 112852, 111362, 48780]],
```

Appendix 341 - samplesize and predictions

uid	sample	recommended movies
113750	1000000	[115617, 135143, 39183, 2353]
113750	2000000	[112852, 128360, 59315, 2353, 5574]
113750	3000000	[2028, 59315, 59392, 2353, 115664, 96588]
113750	4000000	[81845, 134853, 96588, 152591, 81847, 2353, 77561, 59315, 115664, 59392]
113750	5000000	[122882, 3000, 135143, 134853, 59315, 5690, 93510, 59392, 6874, 6365]
113750	6000000	[134853, 44191, 73017, 33794, 77561, 6377, 59784, 59315, 6333, 148626]

Appendix 342 - Prediction test 1 Hit rate how many of the testset is accurately picked

```
print(recommendedmovies[0])
print(truepreference[0])
```

```
[101382, [858, 1221, 527, 953, 356, 1387, 608, 1304, 1267, 1210]]
[101382, [1210, 364, 1263, 527, 1276, 356, 1302, 858, 508, 1304]]
```

```
def top_n_accuracy(recommendedmovies,truepreference):

    uid = []
    score = []
    a = 0
    for i in recommendedmovies:
        x = 0
        y = 0

        for j in i[1]:
            if j in truepreference[a][1]:
                x += 1
                y += 1
            else:
                y += 1

        acc = x/y
        a += 1
        uid.append(i[0])
        score.append(acc)

    return [sum(score)/len(score),uid,score]
```

```
prediction_score = top_n_accuracy(recommendedmovies,truepreference)
```

```
prediction_score[0]
```

```
0.7807301336442153
```

Appendix 343 - prediction test 2 Hit rate how many movies from the model is in the top rated of 1 user

In [97]:
```
all_predict = []
for i in movieids:
        all_predict.append(svd.predict(101382, i))
```

In [132]:
```
user101382movie5results
```

Out[132]:
```
defaultdict(list,
            {101382: [(858, 5),
              (1221, 5),
              (1097, 5),
              (260, 4.977891434033649),
              (527, 4.918608089054963),
              (2028, 4.909324110426823),
              (912, 4.905609121316953),
              (1193, 4.898487125892857),
              (111, 4.884755531562192),
              (1196, 4.861945850044397)]})
```

In [148]:
```
len(user101382movie5reallist)
```

Out[148]: 48

In [144]:
```
score = []
for i in user101382movie5resultslist[0][1]:
    x = 0
    y = 0
    if i in user101382movie5reallist:
        x += 1
        y += 1
    else:
        y += 1

    acc = x/y
    score.append(acc)

print(sum(score)/len(score))
```

```
0.8
```

In [164]:
```
all_predict = []
for i in movieids:
        all_predict.append(svd.predict(270893, i))

user270893movie5results = get_top_n(all_predict,n=10)
user270893movie5real = ratings[ratings['userId'] == 270893][ratings['rating'] == 5
user270893movie5reallist = user270893movie5real.to_list()
user270893movie5resultslist = []
# Append the recommended items for each user
for uid, user_ratings in user270893movie5results.items():
    user270893movie5resultslist.append([uid, [iid for (iid, _) in user_ratings]])

score = []
for i in user270893movie5resultslist[0][1]:
    x = 0
    y = 0
    if i in user270893movie5reallist:
        x += 1
        y += 1
    else:
        y += 1

    acc = x/y
    score.append(acc)

print(sum(score)/len(score))
```

```
0.4
```

Appendix 344 – Prediction test 3 Hit rate the predicted movie list is not in the trainset

In [62]:
```
all_predict = []
username = 270893

movies_watched = []
for r in trainset.all_ratings():
  if r[0] == trainset.to_inner_uid(username):
    movies_watched.append(trainset.to_raw_iid(r[1]))

movies_watched_all = list(ratings[ratings['userId'] == username]['movieId'])

movies_watched_later = []
for i in testset:
    if i[0] == username:
        movies_watched_later.append(i[1])

movie_not_watched = list(movieids)

for i in movies_watched:
    movie_not_watched.remove(i)

for i in movie_not_watched:
    all_predict.append(svd.predict(username, i))


user_movie_results = get_top_n(all_predict,n=10)
user_movie_results_list = []

# Append the recommended items for each user
for uid, user_ratings in user_movie_results.items():
    user_movie_results_list.append([uid, [iid for (iid, _) in user_ratings]])

score = []
for i in user_movie_results_list[0][1]:
    x = 0
    y = 0
    if i in movies_watched_later:
        x += 1
        y += 1
    else:
        y += 1

    acc = x/y
    score.append(acc)

print(sum(score)/len(score))
```

```
0.0
```

In [75]:
```
usernamelist = [101382,70284,69691]
for username in usernamelist:
    all_predict = []

    movies_watched = []
    for r in trainset.all_ratings():
      if r[0] == trainset.to_inner_uid(username):
        movies_watched.append(trainset.to_raw_iid(r[1]))

    movies_watched_all = list(ratings[ratings['userId'] == username]['movieId'])

    movies_watched_later = []
    for i in testset:
        if i[0] == username:
            movies_watched_later.append(i[1])

    movie_not_watched = list(movieids)

    for i in movies_watched:
        movie_not_watched.remove(i)

    for i in movie_not_watched:
        all_predict.append(svd.predict(username, i))


    user_movie_results = get_top_n(all_predict,n=10)
    user_movie_results_list = []

    # Append the recommended items for each user
    for uid, user_ratings in user_movie_results.items():
        user_movie_results_list.append([uid, [iid for (iid, _) in user_ratings]])

    score = []
    for i in user_movie_results_list[0][1]:
        x = 0
        y = 0
        if i in movies_watched_later:
            x += 1
            y += 1
        else:
            y += 1

        acc = x/y
        score.append(acc)

    print(username, " = ", sum(score)/len(score))
```

```
101382  =  0.4
70284  =  0.0
69691  =  0.2
```

```
usernamelist = list(userids)
usernamelist = usernamelist[:100]
combinedscore = []
errorlist = []

all_inner_id = []
for r in trainset.all_users():
    all_inner_id.append(r)

all_raw_id = []
for r in all_inner_id:
    all_raw_id.append(trainset.to_raw_uid(r))

for username in usernamelist:
    if username not in all_raw_id: #error handler
        errorlist.append(username)
    elif username in all_raw_id:
        all_predict = []

        movies_watched = []
        for r in trainset.all_ratings():
            if r[0] == trainset.to_inner_uid(username):
                movies_watched.append(trainset.to_raw_iid(r[1]))

        movies_watched_all = list(ratings[ratings['userId'] == username]['movieId'])

        movies_watched_later = []
        for i in testset:
            if i[0] == username:
                movies_watched_later.append(i[1])

        movie_not_watched = list(movieids)

        for i in movies_watched:
            movie_not_watched.remove(i)

        for i in movie_not_watched:
            all_predict.append(svd.predict(username, i))
```

```
        user_movie_results = get_top_n(all_predict,n=10)
        user_movie_results_list = []

        # Append the recommended items for each user
        for uid, user_ratings in user_movie_results.items():
            user_movie_results_list.append([uid, [iid for (iid, _) in user_ratings]])

        score = []
        for i in user_movie_results_list[0][1]:
            x = 0
            y = 0
            if i in movies_watched_later:
                x += 1
                y += 1
            else:
                y += 1

            acc = x/y
            score.append(acc)
        combinedscore.append(sum(score)/len(score))

print(sum(combinedscore)/len(combinedscore))
```

```
0.08686868686868685
```