Team Members:
Nirmith D'Almeida (#101160124)
Johnathan Scaife (#101145480)
Ali Hassan Sharif (#101142782)

COMP 4601 A2 - Analysis of Different Recommender Systems

Abstract:

An industry revolutionising feature across software services have been recommender systems. You may have seen these systems in places like Amazon, Netflix, YouTube, and more increasingly, Social Media. The idea is to provide content which customers are more likely to consume by analysing consumer habits such as ratings and comparing a user's habits to a similar customer's consumer habits. If we determine two different users generally have the same taste, we can predict if user A will respond to product X if a "similar" user B is recorded to respond well to product X. This is the user-based recommender system in a nutshell. There is also an item-based recommender system, which analyses content that is being engaged with by a certain user, and then recommending "similar" content, which is done by comparing the similarity between the two contents.

The premise is, by recording what consumers respond to, calculating similarity values between responded content and non-responsed to content, we can estimate their responsiveness to the non-responded to content.

By delivering content which has a high likelihood of engaging a user, we massively increase user satisfaction and engagement, which consequently improves the value of your services just as much. The sheer business value provided is significant, and a robust recommender system is the key in seeing that value increase.

Our team, Nirmith, Ali, and John, analysed 2 popular recommender system algorithms, item-based recommendations and user-based recommendations. Think of the item-based approach as estimating the similarity between content users will consume, and user-based as estimating the similarity between users based on the content they consume.

Our dataset was a 2D matrix, each row corresponds to a specific user, and each column corresponds to a specific item. An element in the matrix is the rating value a specific user gives to a specific item, similar to a review you can give for an Amazon product.

In this paper, we will describe how we determine the accuracy/competence of a recommender system, a breakdown of the two types of recommender systems analysed, and hyper parameters we fine tuned throughout our experiments. In the end, we hope to provide insights into what makes a recommender system perform at its highest accuracy.

Key Algorithms Implemented

User-Based Recommender:

1. filterReviews(userA: Array{int}):

   This function takes in a 1d array of integers, userA.

   The values of this array are the reviews the user has given to product x, where x is an arbitrary index of the array. We filter the review array to only include items where a review is recorded for userA.

   It returns the filtered array.

2. findNeighbours(userData: Array{ Array{int} }, userIndex: int):

   This function takes in a 2d array, userData, which represents all of our user data, as well as an integer userIndex, which is an index in our userData. This index in our userData corresponds to a specific user. Note that userData is of size n x m, where n is the number of users, and m is the number of products.

   findNeighbours will formulate the “neighbours” of userA, which is measured by a similarity metric between userA and userB, where userB is all other users in our userData. Our neighbours are represented by a 2d array of dimensions (n - 1) x 2. We have n-1 neighbours, and 2 values to represent each neighbour, a similarity metric, and the index to reference the neighbour in userData. After our neighbours array is constructed, we sort the neighbours in descending order by their similarity value. This means the user most similar to userA will be neighbours[0], and the user lease similar to userA will be neighbours[n].

   We return our formulated neighbours array after calculations.

Description of findNeighbours

Evaluating similarity between users stemmed from the pearson correlation coefficient. We would take the set of reviews users have given to a set of items, and evaluate how similar their reviews are. The more correlated, the more similar.

findNeighbours takes in the review data for all users, as well as the second parameter userIndex, which will be the user we are trying to find neighbours for. After applying pearson correlation to all users (except for the user itself), we spit out an array of all correlation coefficients for each user, and the user index stays attached to each value of the array. We sort the array in descending order by similarity, and return this array.

This function is key to the recommender system, as it describes the similarity values needed to make our predicted recommended score later on.

3. findNeighboursTLess(userData: Array{ Array{int} }, userIndex: int, threshold: int):

This function takes in the user data once again, as well as an index to refer to the user whom we are calculating similarities for. We also have a third parameter, threshold.

The first step is to call findNeighbours with the first two parameters to receive our neighbour similarity list.

Then, we filter the similarity array for all similar neighbours that have a similarity value less than the given threshold. This means if the given threshold is 0.5, any user who has a similarity correlation coefficient greater than 0.5 is ignored.

4. findNeighboursTGreater(userData: Array{ Array{int} }, userIndex: int, threshold: int):

Same behaviour as findNeighboursTLess, but we will take similar neighbours that are greater than the given threshold, instead of less.

5. leaveOneOut(userData: Array{ Array{int} }, settings: String, parameter: Number):

leaveOneOut is the evaluator for our recommender system. It takes our userData array, which represents all reviews given by each user for all items. A value of 0 means no review exists between the user and that item.

The second parameter String is allowed 3 different string values. The first one - “topK”, second - “t-above”, and third - “t-below”. These are signals to the leaveOneOut evaluator if we will calculate neighbours based on top K similar neighbours, all neighbours above a given threshold, or all neighbours below a certain threshold.

The third parameter is either a threshold number between -1 to 1, or a k value, which is an integer greater than 0. The settings parameter will determine how leaveOneOut treats the third parameter.

This function will proceed with a “leave one out” style of accuracy evaluation. We will iterate through each and every review given by each and every user, a total of n * m iterations. For each review x given by user A, we will remove that review x from user A’s data, and create a temporary userData copy with the exception of the review x we have taken out.

Now, we will use a variation of findNeighbours to calculate our similarities for the given approach. This is defined by the settings and parameter variables.

Once we have our similar users to predict off of, we will create a prediction value in place of the review x we have removed. The prediction method is as follows:

Predicting User Review for item X from Similar Users' Review of item X

There are two parts to the prediction of a review. First, we take what user A's average review value is, the same as a rating A would give if he were neutral for a particular item. Let's say A has given 4 reviews of values 5, 1, 5, 1. The average of A's reviews is 3, which is where the initial value of our prediction starts.

Then, for each similar neighbour, we calculate their average review as well. Let's say user B has an average review of 2.

The next part is crucial to understand. If B gives a rating of 3 to an item, and A gives a rating of 3 to the same item, it is incorrect to assume they responded to the item in the same manner. Since A gives out higher ratings more often, 3 is a normal experience of satisfaction. However, user B's review of 3 is 50% greater than his usual review of 2. This shows B is slightly delighted by the item. Although the rating value is the same, the "satisfaction" is determined by the deviance from the average rating.

With this in mind, we loop through all neighbours, and see their "satisfaction experience" in relation to their average review. The deviance is the key metric to predict off of. We take this deviance from average, whether it is below or greater than their normal review, and use that to calculate the deviance user A will have from the initial prediction of A's average review. Keep in mind, we will be using the similarity metric to weight the deviance accordingly as well. This means, a neighbour with similarity of -1 and a deviance of -2 from their average rating for item X will result in a prediction of +2 deviance from A's average review of 3.

We perform this deviance calculation and similarity weighting for all neighbours and are left with a final prediction for item x.

Mean Absolute Error (MAE) Loss - Recommender Evaluation

After we have our prediction calculated, we can calculate how accurate it is compared to the actual review we removed at the beginning of the iteration from the loss observed from MAE. If the loss is 0, that means the prediction was completely accurate. The greater the loss, the less accurate our prediction will be.

Final Evaluation

We repeat **leaveOneOut** evaluation n * m times, and observe the average MAE loss. The final MAE result represents the deviation we normally have from the actual rating a user A gave compared to what we would predict user A to give. For instance, if the MAE is 0.75 and the prediction we came up with is 4 for a given A reviewing X, on expectation, the actual review is most likely between 3.25 - 4.75. There will be predictions that are completely accurate and predictions that might make no sense. But on average, we see a 0.75 deviation from the actual prediction.

<u>Item Based Algorithm</u>:

This algorithm is a recommendation system technique that unlike User based works by analysing the relationship between items themselves. The algorithm is based on their similarity to other items that they interacted with/rated.
Functions and formulas used:
We use the cosine similarity and prediction based of items rated that are near neighbours to that product from the slides (refer Figure)

$$sim(a,b) = \frac{\sum_{u \in U}(r_{u,a} - \overline{r_u})(r_{u,b} - \overline{r_u})}{\sqrt{\sum_{u \in U}(r_{u,a} - \overline{r_u})^2}\sqrt{\sum_{u \in U}(r_{u,b} - \overline{r_u})^2}}$$

Where *U* is set of users that have rated both of a/b

$$pred(u,p) = \frac{\sum_{i \in ratedItems(u)} sim(i,p) * r_{u,i}}{\sum_{i \in ratedItems(u)} sim(i,p)}$$

***ratedItems(u)*: items u has rated that are nearest neighbours to p**

Figure : Prediction and cosine similarity formulas

Our implementation in detail:

- We take the file and convert and strip it to a nested matrix called ratings of size N x M. We generate a corresponding zero matrix of the same size and we update for wherever the rating is ! = 0 with the new predicted value and the corresponding zero value remains there as it is.
- Here we calculate our Averages for the user UP AND UNTIL that specific rating and we then set average to 0 if the number of valid ratings is 0. After we have this setup we calculate cosine similarity for Item Index i and Item Index j and we return it for that value (please do remember this is between items so we ensure that the 2 item indexes are not the same otherwise it will be a 0).
- We then predict the rating based on the similarities and settings (topK or threshold). For both we calculate the ratedSimilarity and verify if the ratedSimilarity skips over negative similarities and they are not the same item.
- From here we notice a few edge cases arising: if there is similarity we just return averageForThatUser, we check if the neighbourhood size is > than our rated similarity collection size if it is we reset this to our new rated similarity collection size. And we finally sort and take top size (updated or same) values and calculate the prediction for this using the formula for prediction above. We then set our limits to prevent >5 and <=0 values to 0 and 1 correspondingly. After we calculate our prediction we reach the final step of our calculations for MAE (Mean Average Error). Which is the absolute\ difference between our prediction for that ith and jth index (User i and Item j) and our original rating and return the sum of these values.

How have we minimized our run time complexity for Item Based:

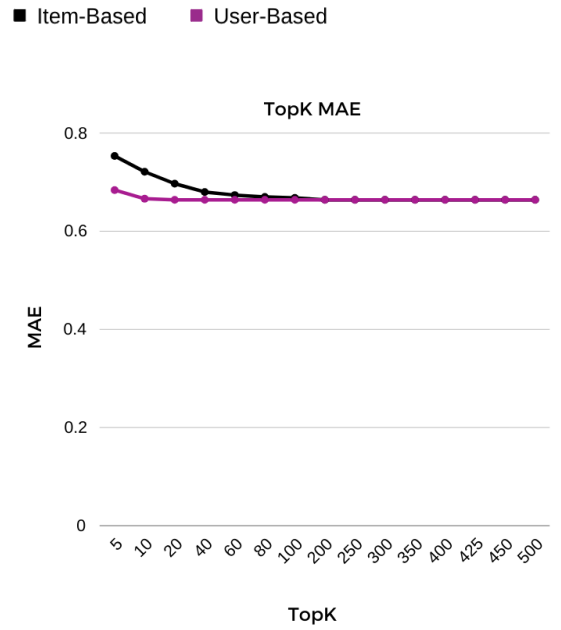
- We managed to cut some time with the way we were calculating our averages, total sums and numbers for valid ratings and then updating it when we would go through the predictions and findNeighbour implementations. The Next step we did to cut off our time complexity was to rework the pre-computing of similarities to being calculated within.
- The Other Solution we noticed was implementing this with using libraries and modules that auto computes and calculates everything for us for instance mlMatrix, pre computing and using library and modifying data to make sure we reach the end goal.

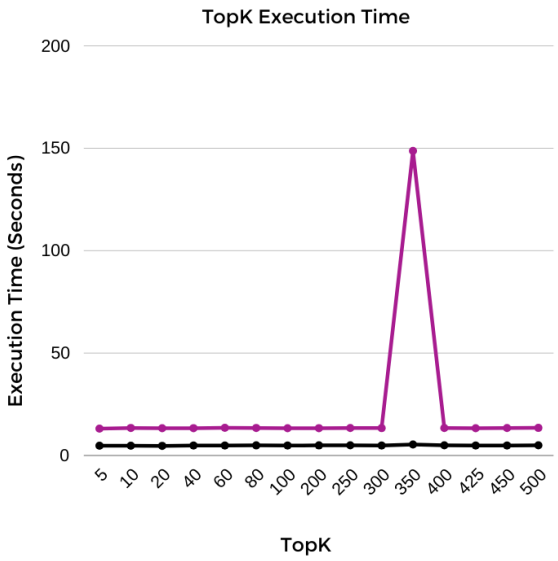
Our Findings:

After implementing both the item-based and user-based algorithms for TopK and Threshold. We began running tests to determine the MAE, and running time. Our goal was to determine which method provided the most accurate data, in the least amount of time. Our data is as follows:

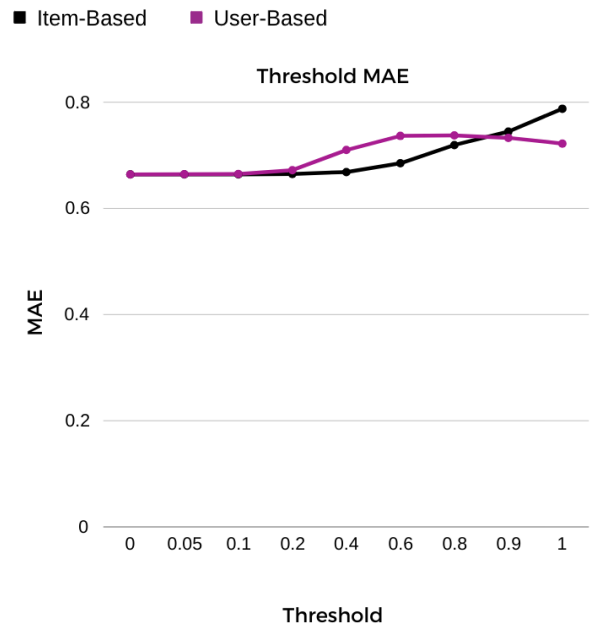
DATA:

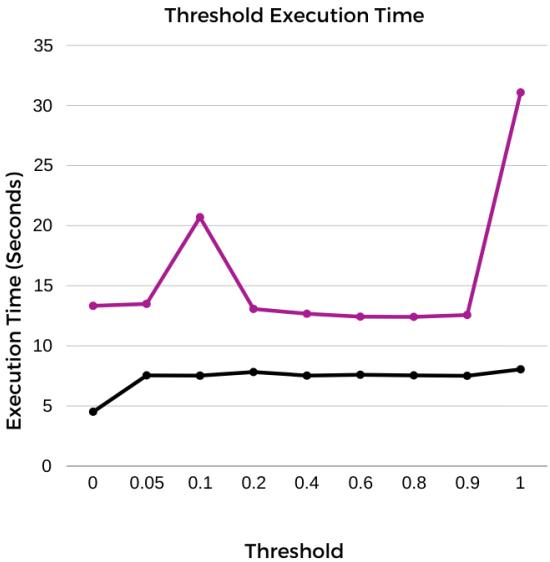
TopK





Threshold





We can deduce a few key points from this data…

1) What algorithm is most accurate

From the data we can determine that given the correct data, any of the algorithms could hold an MAE of about 0.66. However, that is only if the correct conditions are met.

TopK - TopK's data indicates that both the item-based and user-based models produce an MAE around 0.66 at around 200 neighbours. The MAE slowly becomes worse anywhere below that number of neighbours. However, the item-based model seems to regress faster than the user-based model. At a TopK of 5, item-based reaches an MAE of around 0.75, while user-based reaches an MAE of just 0.68. Intuitively, these numbers make sense as with a higher TopK, we compare more neighbours, and therefore have a more reliable sample size to predict ratings.

Threshold - Similarly to TopK's data, the item-based and user-based models produce an MAE around 0.66 with a threshold lower than about 0.1. This MAEs progressively gets worse as this threshold increases, but the pattern at which the MAE regresses is slightly different than in TopK's case. The item-based model's MAE steadily gets worse, however the user-based model has a sharp jump before levelling out. This might indicate that users are separated into very secular cliques. Once our threshold allows users from other cliques, with much less overlapping interests, there is a sharp dropoff in the MAE.

2) How long does the predictions take?

Our data appears to have some outliers in the time it takes to complete the predictions. This is most likely due to the cpu of the computer, and not an indication of an extreme jump in program complexity. Overall, the running time is very steady for both TopK, and Threshold. This is most likely due to the smaller data set we are using. However, in both cases the user-based running time is higher than the item-based running time. This is most likely due to there being around 10x more items than there are users. Since we put a cap on the number of users/items we are comparing. The running time mostly lies in calculating the similarity between users/items, which is bounded by the number of possible ratings, rather than calculating the actual prediction, which is bounded by the number of other users/items we are comparing with. The amount of users that can rate an item is much smaller than the number of items a user can rate. Besides that, the running time is overall slightly longer, and more unstable in threshold based, rather than TopK based. This is most likely because the number of similar items isn't bound by a hard limit like in TopK. Theoretically there can be thousands of similar items within a specific threshold, but TopK selects on the top k of those.

3) What algorithm/parameter would we use for a real-time online movie recommendation system?

Well it sort of depends on the expectations of the number of users, and what you value.
If we are expecting to have more users than there are movies, it would probably be best to use user-based. As mentioned, the majority cost of the predictions is based on the number of potential ratings the user can make, or the number of potential ratings a movie can have. If we have millions of users, then it would be much less costly to get the similarity between users based on their ratings of a thousand movies compared to getting the similarity between movies. Based on a million users. As for the parameters, I would most likely go with TopK. The running time for TopK is very consistent, and is influenced less by the number of users/movies compared to the threshold. Once again, if we are providing a service for a million users, a threshold of 0.001 could produce thousands of users. Whereas with TopK on a billion users, it would still only compare K users. We just need to ensure we have a large enough sample size, and then we will get an accurate prediction. Just comparing 200+ similar users should be enough to get accurate predictions, no need to calculate more than that.

4) How will our solution be affected by more/less reviews

It wouldn't affect the running time, since that is more based on the number of users and the number of movies. However, it would certainly affect the accuracy of the predictions. If we did not have many reviews, TopK would be taking neighbours which don't have high similarities because there is no way to tell if neighbours are similar due to the lack of reviews. Threshold method will also suffer, as the number of users/items that fall within the threshold will be extremely small. Leaving the results easily influenced by only a few neighbours.