USING ANT COLONY OPTIMIZATION AND GENETIC ALGORITHMS AS THE BASIS FOR A FILM RECOMMENDATION SYSTEM.

Institution: Carleton University

Author: Sepehr Eslami Amirabadi

STD-ID:101112474

Supervisor: Prof. Ahmed El-Roby

Dec 7, 2022

ABSTRACT

With the ever-increasing production of new movies and with modern technology providing access to a vast library of options, it becomes increasingly relevant to have accurate recommendation systems to prevent users from experiencing choice paralysis. Humanity is incredibly diverse and while this is normally seen as an advantage, it makes the creation of an accurate recommendation system difficult. In this study we combine the strengths of Ant Colony Optimization with the strengths of the Genetic Algorithm to produce a basic recommendation system that produces a list of accurate recommendations. It is shown in the paper that the Genetic Algorithm does in fact improve the performance of a system that is solely trained with ACO. The results also indicate that users can successfully ask for recommendations using this system with genres as query parameters.

ACKNOWLEDGMENTS

I would like to acknowledge the vast contributions of Harper and Konstan as well as anyone else who contributed to the MovieLens DataSet found at https://grouplens.org/. Their Dataset was nicely organized and easy to use. Without them this paper would not have been possible. I would also like to thank my Project Supervisor, Ahmed El-Roby for taking the time to be my supervisor and provide feedback on my work.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

MOTIVATIONS AND BACKGROUND INFO

There are many forms of media that we as a society consume for entertainment. Whether it's reading a good book, watching a fun movie, or playing an immersive game, we as a species have devised incredibly creative and entertaining methods for spending our free time. Sometimes however, the enormous variety of films and other media can be incredibly overwhelming and leave us with choice paralysis. This burden of choice is something that can be reduced and perhaps even removed entirely with the implementation of a suitable recommendation algorithm.

This is easier said than done however as the preferences of an individual are deeply personal and complex. With dozens of macro genres and hundreds of sub genres in cinema, it is evident that there will be some complex user preferences that will be hard to account for in a recommendation algorithm. In this Paper I plan to make use of Ant Colony Optimization (ACO) techniques in conjunction with the genetic algorithm to build a movie recommendation system.

Ant Colony Optimization (ACO) is a machine learning method often used to solve the traveling salesman problem. The traveling salesman problem is a classic shortest path problem, where we try to determine the ideal order of visiting all the cities on a graph in a way that minimizes total distance traveled. ACO is based off the real-life behaviours of ant colonies; ants will randomly choose a certain path and travel that path; while traveling they release pheromones on the path that fades over time. An ant is not likely to repeat a path that was long and will therefore release fewer pheromones on that path. Future ants are more likely to travel paths with more "pheromones," therefore overtime an ideal path converges which all the ants will use.[Dorigo, et al., 2006] In our study we will be leveraging the behaviour of Ant colonies to develop a movie recommendation algorithm. ACO is typically used to optimize a parameter while traveling to every node in the graph. In the traveling salesman problem, we use ACO to optimize path length, in our use case we will optimize the average movie ratings/enjoyment of a path.

A Genetic Algorithm (GA) is an algorithm that is based off the theory of evolution by Charles Darwin. Typically, a genetic algorithm starts with a set of agents that will achieve a certain task. How well they do at said task will determine their "fitness". Later, these agents will copulate and produce a new generation of agents, the new generation is determined by the fitness of their parents and a random "mutation" function.[Lingaraj, 2016] After x number of generations of "agents" you can stop training and if configured properly your current generation of agents should be much better at whatever "task" they are assigned than the starting population. In this study the genetic algorithm will be used to "cull" and evolve the "ants" (other users) to both augment the dataset by creating new "ants" with more diverse movie preferences and as a means of introducing a random factor to movie recommendations due to "mutations."

Ant colony optimization is a very versatile machine learning technique that has been used to solve a variety of different problems. A Movie recommendation system using ACO and artificial bee optimization[Sethi and Singhal, 2017]. A Student Course Recommendation system [Sobecki and Tomczak, 2010]. The genetic algorithm has also been leveraged to solve similar problems. The creation of a popularity based news recommendations system using GA [Manoharan et al., 2022]. A genetic algorithm-based query recommendation system for search engines [Barman et al., 2020]. GA has even proven to be effective in solving the reliability-redundancy problem for online systems [Kim and Kim, 2016]. It is clear that there is precedent proving both the efficacy of GA and ACO for recommendation systems, in this study I will explore the efficacy of using the two in conjunction.

METHODOLOGY

ANT COLONY OPTIMIZATION STRUCTURE

The Ant Colony Optimization program developed for this Project makes use of the following classes:

Ant Module: This class represents the “agents” that will be traversing the graph.

Class variables:

- Trail: represents the “movies” the ant has “visited”/ watched while traversing the graph
- Current movie: represents the last movie that the ant has “seen”
- Review list: keeps track of the ants’ reviews for movies. Used for pheromone updating

Member functions:

- get_cost: will return the ants review for a specific movie
- Watch_movie: updates the ants trail

Ant Colony: This class represents a colony of ants. It is responsible for moving ants to their starting node as well as prompting the ants to move to the next node on their "journey".

Class variables:

Ants: an array of ant objects

Member functions:

Determine next movie: takes an ant and graph weights as input and finds the movies the ant will want to watch next

Move all ants: moves all ants to the "best" movie for them using "determine next movie" member function

Ant Graph: This Class executes the ACO algorithm.

Class variables:

Ant colony: a colony of ants

Vertices: vertex of the "graph" represents movieID's

Graph_edge_weights: represents the desirability of watching a movie based on pheromones dropped on edge and personal rating of movie

Num ants: represents the number of agents we want traversing the graph

Num movies: number of movies the user wants recommended

Member functions:

One_iter: makes the appropriate calls to AntColony to complete one tour of the graph

Update_edge_weights: updates the edge weights based on all ants tours. (updates Pheromones)

Task: This class defines the a film and all its details

BASIC ALGORITHM

In preprocessing steps that I will describe later in the methodology, we compute a level of similarity with all the other users in the dataset based on movie selection and ratings. Then in the Genetic algorithm stage, the users' watched movies and ratings are taken, paired off with other similar users, and a "child" is created that is a splicing of the ratings of the two "parents". We run that for several generations and are provided with an "evolved" set of users. These evolved users are then used as input for ACO.

Each user that the system wishes to provide recommendations for, will provide a subset of their movie history, the number of agents/ants they want and the number of iterations to run. Using the user's movie history we will compute levels of similarity between the user and the "Evolved user set", then we will take the n most similar users to act as our "Ants". These ants will travel a graph computed from a list of all the movies that all the "Ants" have seen and drop "pheromones" on the edges. Eventually an optimal trail will emerge traversing the movies in the graph, these will be our recommendations.

It is worth noting that traditionally in ACO ants would visit every node in the graph before the tour is completed. However, in this version of ACO the ants only travel to as many movies as the user asks for. This was done both due to computational constraints and due to a difficulty converging on a solution due to excessive variance.

GENETIC ALGORITHM STRUCTURE

The genetic algorithm which is loosely based on the theory of evolution was performed on user reviews to increase the variety of user input data with the hopes of increasing the predictive accuracy of our model. The GA loosely mimics evolution using the following 4 functions:

1. FITNESS FUNCTION:

This function determines the evolutionary fitness (aka likelihood that the agent passes on their "genes"/movie ratings)

It is computed by finding the mean similarity of each user. We form a similarity vector that rates the similarity between one user and all the other users based on their reviews in the preprocessing steps. We take the mean of this vector and if the avg similarity is in the bottom 25th percentile these users are deemed unfit and will not be included in the next generation. What this will achieve is that the users who can be considered to have "weird" preferences (have the least in common with the other users) will be filtered out. This will increase the uniformity of our data thereby boosting general performance

2. SELECTION:

selecting the most fit and the most similar "users" and pairing them off for mating/passing on their genes.

This function will determine the “couples” which will have children. The equivalent evolutionary concept would be “sexual selection”. This is simply done by pairing together the users that were not deemed unfit with the user most similar to them.

3. CROSSOVER

selecting random components from both agent when creating offspring

In this function we choose the “genes” the child will inherit. Essentially, we choose random indices from one user in the couple and random indices from another user in the couple and splice them together.

4. MUTATION

Account for random variance by adding traits randomly.

This function is intended to add an element of randomness or surprise to the data. We do not want the system to discriminate solely based on previous preferences, we want the system to be able to provide recommendations that are novel and might break the trend. Humans are not entirely deterministic, sometimes we hate movies in genres we like and like movies in genres we hate. This function aims to make our data more representative of the randomness that can shape some preferences.

It operates by randomly choosing some movie from a users review and either increasing the score or decreasing the score. In some cases we add a random movie to a “child’s” list with a random rating.

Together these functions form the genetic algorithm. In this study the genetic algorithm was used to trained 1000 generations of new users. Several different generations were tested on the recommendation system to determine the efficacy of the genetic algorithm. You can look at the python implementation of these four functions in appendix A.

COST AND DESIRABILITY FUNCTION

Traditionally, ACO has a cost for traveling from one node to another. When the cost is lower it is more desirable and the "ant" is more likely to travel across a given edge. The cost is traditionally path length, where the shorter the length the more desirable it is. In our case, the higher the ratings for a movie are, the more desirable an edge becomes.

Cost:

$$\boldsymbol{Cost\ (MID)\ =\ the\ ant's\ rating\ for\ movie\ with\ id\ MID}$$

Desirability: is viewed as a probability of traveling to one node from another

$$\boldsymbol{P(a,b)\ =\ \frac{[\tau ab]^{\alpha}\ [\eta ab]^{\beta}}{\sum[\tau ab]^{\alpha}\ [\eta ab]^{\beta}}}$$

This function represents the probability that an ant agent, will watch movie b after having watched movie a.

$\boldsymbol{\tau ab}$**:** the pheromones on the path (edge weights)

$\boldsymbol{\eta ab}$: $\boldsymbol{the\ cost\ of\ watching\ movie\ b\ (movie\ rating)}$

α = learning rate/ influence rate of pheromone on ant path

β = influence rate of movie rating on ant path

SIMILLARITY MESURE

The similarity function is an integral aspect of the recommendation system. It is used in the Genetic algorithm to determine Selection and coupling, and it is used in ACO to determine which "ants" would traverse the graph when looking for a user recommendation. It was decided to consider 3 separate cases of what can be considered similarity then taking the sum of all three.

Case 1: Number of overlapping movies watched

Using a point-based system we will add 100 points in the similarity metric for every movie that both users have watched

Case 2: rating similarity for movies both users have watched

To properly measure similarity in taste we must also account for differences in opinions on movies two users have both watched. So we will subtract points based on a percentage of difference between the two users.

similarity = 100 * (min(rating1 , rating2) / max (rating1, rating2))

Case 3: overall genre preferences of the users

general genre preferences were also used as a measure of similarity. We consider genre similarity as a dividend of the percentage of each genre the user has watched. 1000 points is the maximum afforded to each genre, similarity will be incremented by the following expression

similarity += 1000 * (% genre user 1/ % genre user 2) * min(% genre user 1/ % genre user 2)

% genre represent the percent of a users movies that fall within a certain genre

After taking the summation of these three cases the values were z-normalized so that they would be bound between 0 and 1. Consult Appendix B if you want to see the code.

DATASET

For the purposes of this study the “ml-latest-small” dataset from the MovieLens database was used. The Dataset consists of 100 000 ratings, 9000 movies, and 600 users.[Harper and Konstan 2015] Initially, I intended to use the “ml-25m” data base which had 25 million ratings, but that was a rather naïve assumption as the computational power needed for such training was beyond our resources.

As a preprocessing step the dataset was restructured so that when querying the dataset with only UserID we would be able to obtain the users entire review list. Then a similarity matrix was computed that contained the user similarity between all users. After Evolving the “users” for 3000 generations using the Genetic Algorithm, the data was prepared for training ACO.

TESTING DATA CONFIGURATION

The goal of a recommendation system is to provide recommendations that a user has not seen before in the hopes that they will enjoy it. The performance of a recommendation system can then be defined as how often a user enjoyed a new movie that was recommended. Since live testers were not possible for the testing of this system we had to get a bit creative.

50 of the 600 users are chosen at random as “test” users. We take 50% of the user reviews of the test users as input for the ACO model, the remaining 50% are used as test data. After running ACO for each user, we are provided with a list of recommendations. The metrics we use to “rate” the accuracy of the recommendations are as follows:

Movie overlap: represents the percent of the recommendations that can also be found in the "test" data. After all, if using half a user's review data, we find movies that the user has already watched, it means that our recommendations picked a movie the user would choose to watch.

Rating Accuracy: If a movie is recommended that exists in the test data, we divide it by 5 to get a percentage of how likely the user was to enjoy the recommendation. For example, if a movie was recommended that the user rated 2 (of 5 stars), then the accuracy of the single recommendation is 2/5 = 0.4.

In our tests we take the Average Movie overlap and the average Rating Accuracy for all test users for several different input configurations.

DIFFERENT TESTING CONFIGURATIONS

Number of Iterations: The number of iterations represents the number of times the "ants" will travel across the graph during training. Tests were run using 10, 30 and 50 iterations to find the ideal number of iterations for convergence to occur.

Number of Agents: The number of Ants that traverse the graph will also affect accuracy. We explored the relationship between increasing the number of "similar users" used to compute the recommendations and the resulting accuracy. Tests were run at 10,20,30,50 agents.

Number of Movies: The number of recommendations will also impact the accuracy. we intuitively expect that the more recommendations a user wants the less accurate they will be. We will explore the performance of ACO for 5,10,20, and 50 movie recommendations.

Number of Generation of the Genetic Algorithm: One of the purposes of this study was to test the efficacy of GA in conjunction with ACO. In these tests we will compare the accuracies at generation 0 (no evolution), 50, 100, 500, and 1000.

Accuracy given a specific genre: One feature that would be useful for recommendation systems is filtering by genre. We will test the accuracy of ACO for a variety of different genres.

RESULTS

OPTIMAL NUMBER OF AGENTS

When finding the average rating accuracy and percent overlap of the recommendation system given by sequentially increasing the number of agents, we can observe the following trends in table 1 and figure 1 below. These numbers are the average metrics across 3 tests for every test condition (i.e. 3 tests per change in num agents). As the number of "ants" or agents is increased there is also a slight but evident decrease in the rating accuracy and an increase in the percent overlap. The implication is that the greater the number of agents we use the more likely we are to find movies that match the users “general interest”, but this decreases the chances of the user enjoying the recommendations. This is an expected outcome. When we use more agents that share similarity with the user, we can expect the results of the “rating/trail” optimization performed during ACO to be more representative of the average interest of the user because the increase in agents will encourage more uniformity. Conversely, since we are taking the top 50 most similar agents rather than the top 10, the agents are on average less similar to the tastes of the user when the number of agents is larger. Essentially, we lose specificity of preferences as we increase the number of agents thereby lowering rating accuracy.

Although one might think that a balance between these two metrics would be ideal, in all future tests we used num agents = 50 because despite the decrease in accuracy, the results are more trustworthy. When percent overlap is low, fewer movies are available to compute the rating accuracy thereby reducing the trustworthiness of the results. This occurs because outliers would have a much larger impact when the sample size is small.

Figure 1

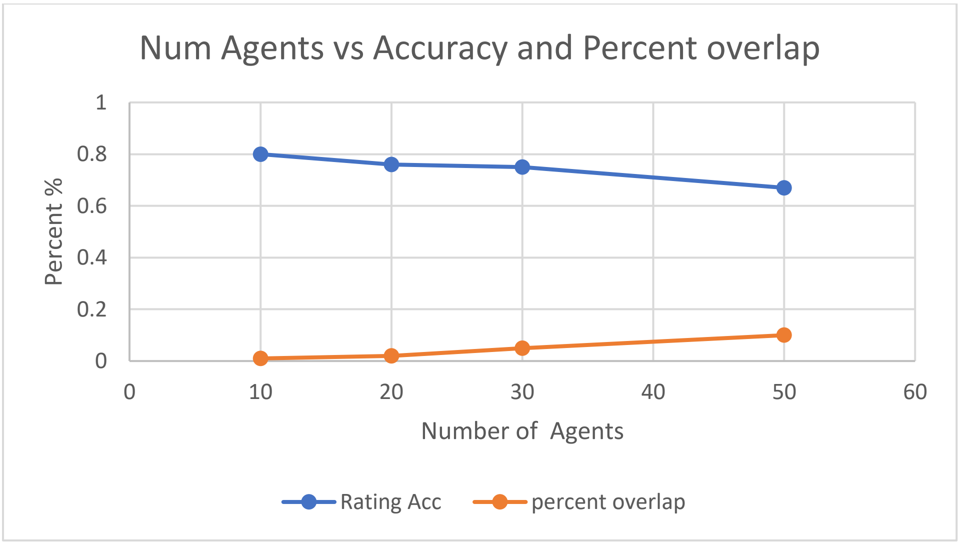


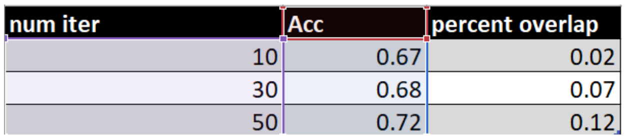
Table 1: Num agents vs acc 1

Num Agents	Rating Acc	percent overlap
10	0.8	0.01
20	0.76	0.02
30	0.75	0.05
50	0.67	0.1

OPTIMAL TRAINING ITERATIONS

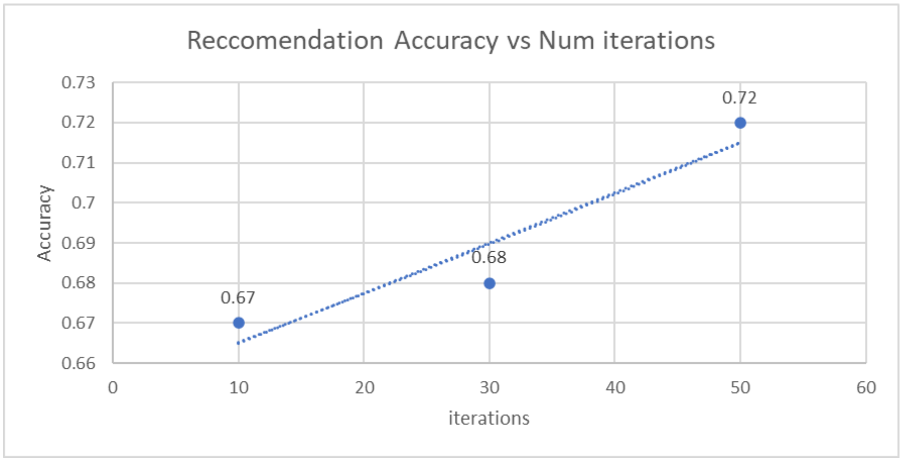
The tables and figure below illustrate the results of tuning the number of iterations as a hyper parameter.

Table 2 Num iterations vs accuracy



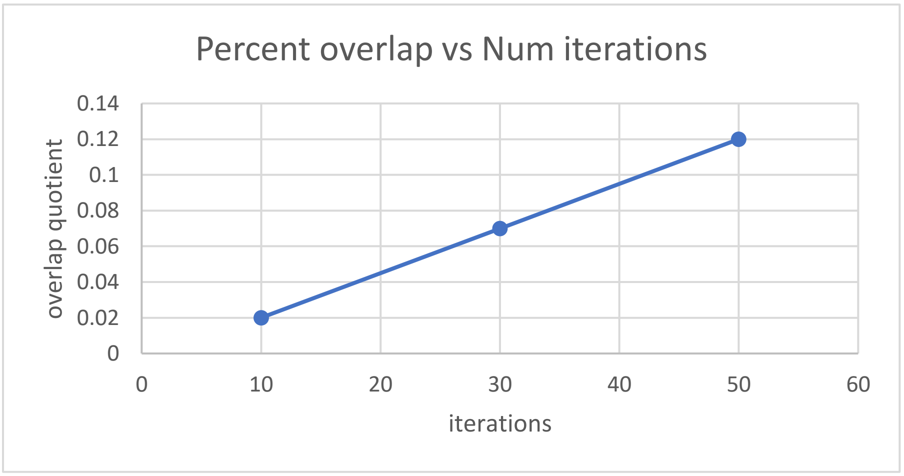
num iter	Acc	percent overlap
10	0.67	0.02
30	0.68	0.07
50	0.72	0.12

Figure 2



We notice that increasing the number of iterations has a positive correlation with the rating accuracy (ie, the likelihood the user will enjoy the recommendations increases as we train for more iterations)

Figure 3

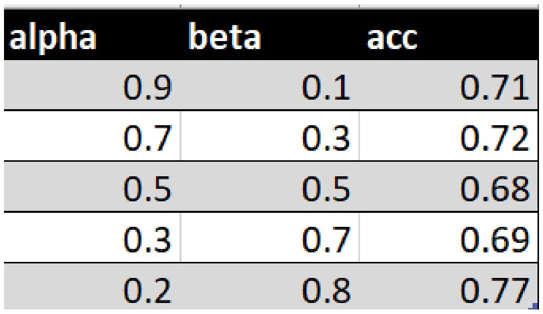


The number of iterations is also positively correlated with the percent overlap. Colloquially this means that the model's ability to provide movies the user would watch but not necessarily enjoy has increased.

The results of these set of tests have no real significance as they were simply used to tune the hyperparameter and find the ideal number of iterations. As with most machine learning models, the more you train the model the higher its accuracy will be. ACO will however eventually converge on a single solution given enough time to train; the pheromones on the “main” trail in the graph will become so high that every ant will take that path. For the purposes of this study due to both time and computational constraints I was unable to run the model enough iterations for complete convergence on a solution. If we had run the model for a sufficiently large number of iterations, we would notice a decrease in the efficacy of each training iteration until, eventually future iterations would have no impact on the metrics at all. Therefore, the ideal number of iterations to use would be the smallest iteration number after the model has converged and the metrics have plateaued.

HYPER PARAMATER TUNEING FOR ALPHA AND BETA

Table 3:



alpha	beta	acc
0.9	0.1	0.71
0.7	0.3	0.72
0.5	0.5	0.68
0.3	0.7	0.69
0.2	0.8	0.77

A set of tests was run to determine the ideal alpha and beta values for training purposes. As mentioned in the methodology, alpha represents, the learning rate or the impact of the pheromones on an agent’s choices. Beta represents the impact of the agents rating for a movie on their choice. There is nothing much to analyse here. It was simply determined that alpha = 0.2 and beta = 0.8 were the ideal hyperparameter values for our system.

HOW MANY MOVIES CAN WE ACCURATELY PREDICT

This is one of the more significant set of tests when trying to determine the overall usefulness of the recommendation system. Although there are a finite number of movies, the variation in human preference with regards to movies is incredibly diverse. The likelihood, of two people sharing general preferences is quite high. The likelihood of two people having the exact same preferences is nearly non-existent. It follows then that the more movies we try to recommend the lower the accuracy will be. After all, it is likely that you and another person enjoy 5 of the same movies, it is very unlikely that you and that same person enjoy 50 of the same movies. In Table 4 and figures 4 you can observe the performance of the model as we increase the number of recommendations.

Figure 4:

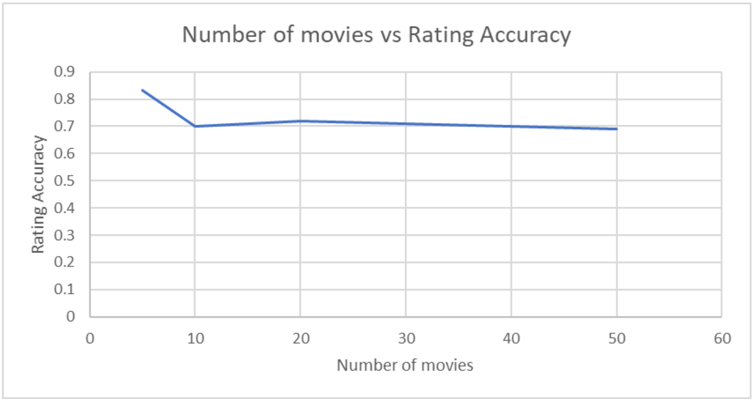


Table 4:

num movies	Rating Acc
5	0.833
10	0.7
20	0.72
50	0.69

As we can see in figure 4, there is a sizable decrease in the accuracy when looking for more than 5 movies. Recommendations for 5 movies has an accuracy of 0.833 while the highest accuracy after 5 movies belongs to 20 movies with rating accuracy 0.72. An unexpected result however was that there is almost no difference in rating accuracy between 10 – 50 movie recommendations. It was expected that as the number of recommendations increased there would either be a linear or exponential decrease in rating accuracy because subsequent movies on the graph would have "weaker" edge weights and therefore be less accurate. The results indicate that the recommendation system would do equally well when recommending 50 movies as it would when recommending 10. The most accurate recommendations of the system however will always be the first 5 movies.

IMPACT OF GENETIC ALGORITHM

The genetic algorithm, loosely based on evolution, is a core component of this recommendation system. Just as evolution helped evolve primates to humans, the genetic algorithm was intended to evolve existing users into superior predictive agents. This was achieved by culling the users that would be poor predictors (see fitness function) and producing "offspring" by coupling the preferences of two similar users (see selection function). This would achieve 2 things simultaneously; it would remove outliers or people with "weird" tastes and it would increase diversity among the "fit"/good users. The following tests were run to determine if the GA had any impact at all on the accuracy of our model. The hyper parameters (alpha = 0.7, beta = 0.3, num_iter = 30, num agents = 50, num movies = 20) were fixed and we simply changed the set of users from which the "agents" were chosen. Generation 0 represents the set of users before running the GA. Generation 50 represents the 50th generation (GA was run 50 times), Generation 100 represents the 100th generation and so on.

Figure 5:

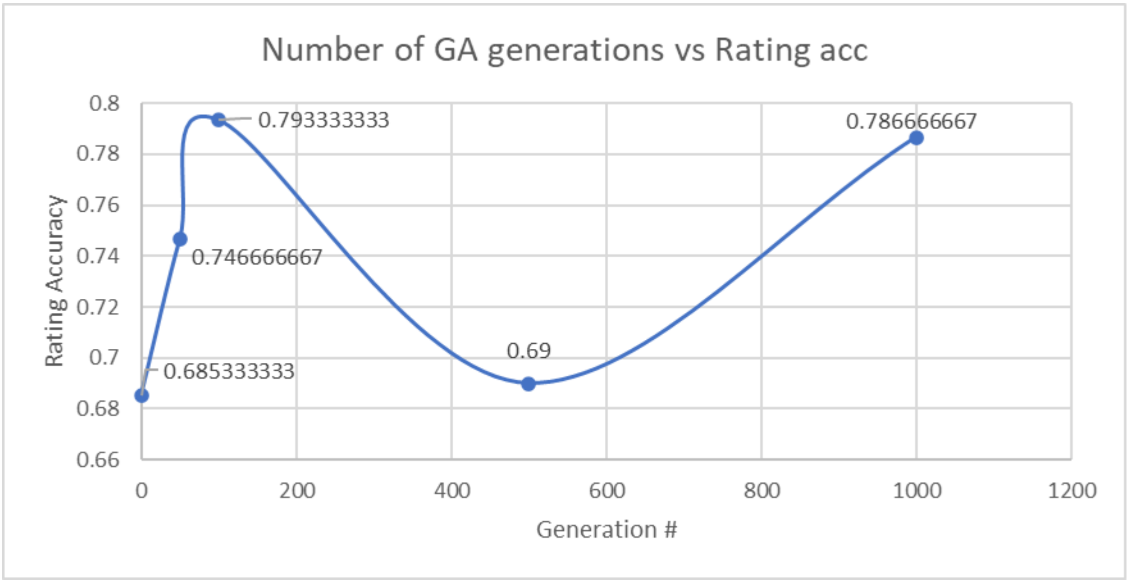


Table 5:

num GA gens (5 movies)	acc_t1	acc_t2	acc_t3	avg acc
0	0.666	0.66	0.73	0.685333
50	0.78	0.76	0.7	0.746667
100	0.8	0.79	0.79	0.793333
500	0.66	0.71	0.7	0.69
1000	0.85	0.79	0.72	0.786667

As illustrated by table 5 and figure 5 above, the genetic algorithm has an undeniable positive impact on rating accuracy. Although there is no distinct correlation visible in the graph, every generation after gen 0 has a substantially higher rating accuracy. Since all cases that went through GA are higher than the original set, this indicates that the genetic algorithm must have had a positive impact.

Most of the rating accuracies follow a positive trend as the number of generations increase, with the distinct exception of generation 500. The dip in variance is unexpected but can perhaps be explained by the somewhat random nature of evolution. Modern apes share a common ancestor with humans, yet we are at the top of the food chain; evolution does not trend in the positive direction where each generation becomes better than the previous, but it trends in the direction of survivability, those who are fit survive and propagate. This is of course purely speculative and will need to be explored further in subsequent studies, but perhaps the generations between 100 and 500 trended in a direction that would increase fitness but decrease its usefulness, much like modern apes.

GENRE SPECIFIC RECOMMENDATIONS

One feature that I decided to implement as part of this recommendation system was the ability to filter and query recommendations based on specific genres. This was achieved by allowing the user to provide a list of genres they either want recommendations for or genres they want excluded. Then the graph that the ants would train over, would remove all movies that do not meet these conditions. I was curious to see how the model would perform over specific genres and whether the specificity of querying would reduce its overall accuracy. Observe the tables and graphs below.

Figure 6:

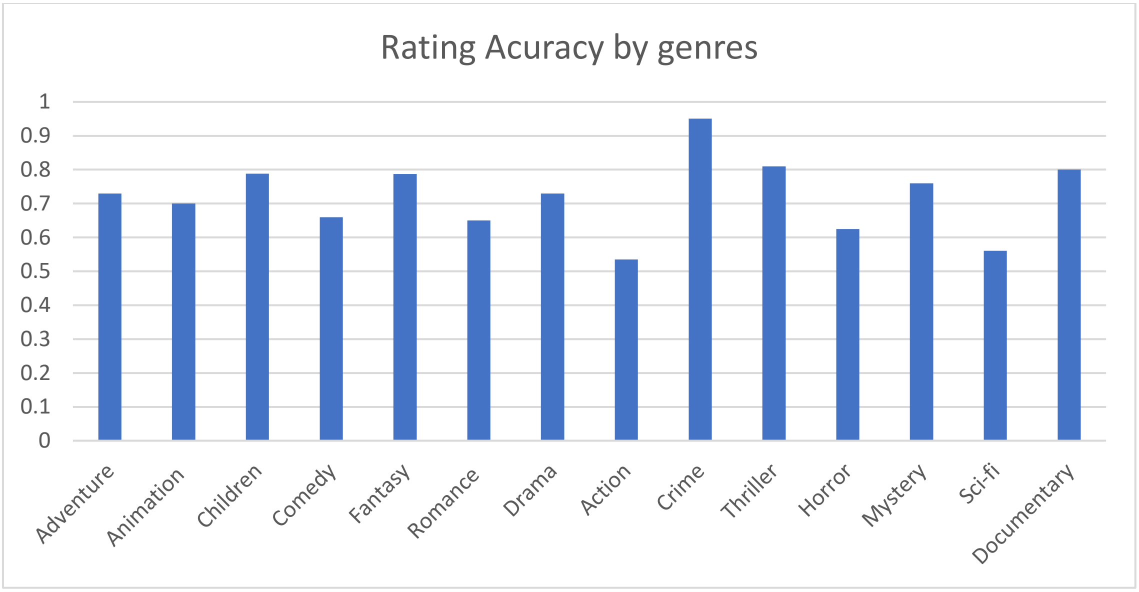


Table 6:

Genres	Rating ACC
Adventure	0.73
Animation	0.7
Children	0.788
Comedy	0.66
Fantasy	0.787
Romance	0.65
Drama	0.73
Action	0.535
Crime	0.951
Thriller	0.81
Horror	0.625
Mystery	0.76
Sci-fi	0.56
Documentary	0.8

As you can see in table 6 and figure 6, these are the accuracies when searching for movies in any of these specific genres. Notably, Action, comedy, horror, romance and sci-fi are the genres that the model struggles to provide recommendations for the most. Crime, Documentary, Fantasy, and Children are the genres that the model is the best at recommending. Overall, the predictions for specific genres are pretty accurate with most of the rating accuracies being above 0.7.

Figure 7:

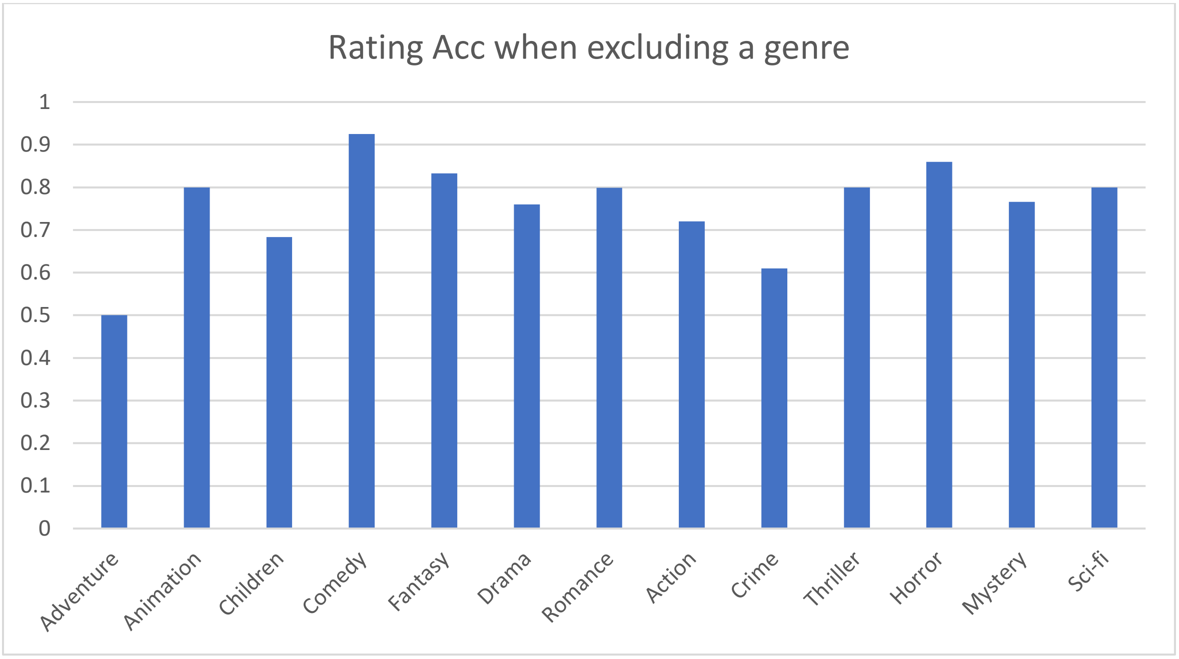


Table 7:

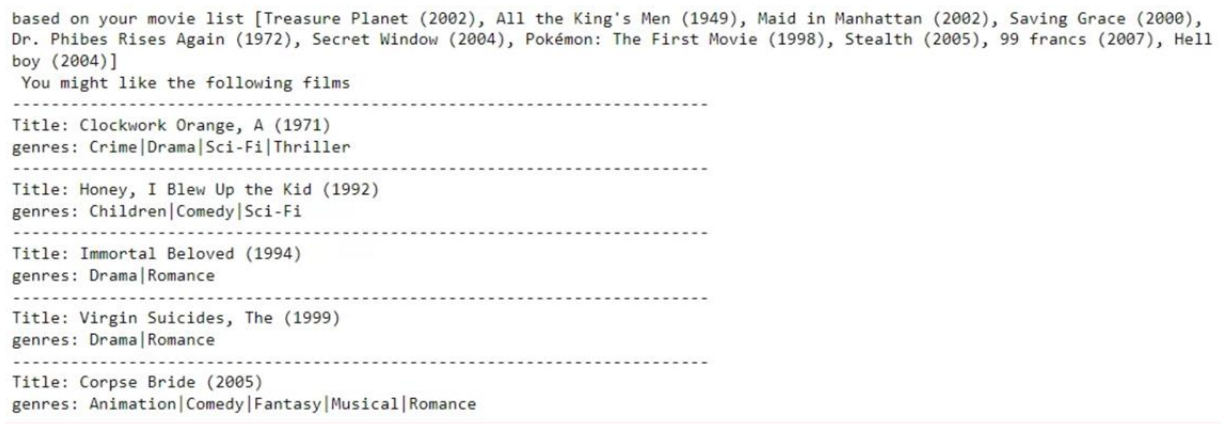
Genres Excluded	Rating Acc
Adventure	0.5
Animation	0.8
Children	0.68333
Comedy	0.925
Fantasy	0.833
Drama	0.76
Romance	0.799
Action	0.72
Crime	0.61
Thriller	0.8
Horror	0.86
Mystery	0.766
Sci-fi	0.7999

Figure 7 and table 7 represent the rating accuracies of the model when we exclude a certain genre. Something notable is that when comedy, horror, and thriller are excluded the overall accuracy is very high (>0.8). These same genres are among the lowest rated in Figure 6. When removed the overall accuracy rises, and when only searching for these genres, their accuracy is very low. This indicates that the model performs poorly for these genres. These genres are a bottleneck on the overall recommendation accuracy.

VISUALIZED RESULTS

When I first decided to explore this topic, I thought that creating a GUI for the system that would display the movie poster and allow the user to fluidly query the system for recommendations would be a great addition. Unfortunately, due to time constraints I was unable to achieve this. Instead, I created a more organized text output that would provide recommendations and details about the movie. Its not very impressive but figure 8 below represents the output of the model for the end user.

Figure 8:



```
based on your movie list [Treasure Planet (2002), All the King's Men (1949), Maid in Manhattan (2002), Saving Grace (2000),
Dr. Phibes Rises Again (1972), Secret Window (2004), Pokémon: The First Movie (1998), Stealth (2005), 99 francs (2007), Hell
boy (2004)]
 You might like the following films
------------------------------------------------------------------------
Title: Clockwork Orange, A (1971)
genres: Crime|Drama|Sci-Fi|Thriller
------------------------------------------------------------------------
Title: Honey, I Blew Up the Kid (1992)
genres: Children|Comedy|Sci-Fi
------------------------------------------------------------------------
Title: Immortal Beloved (1994)
genres: Drama|Romance
------------------------------------------------------------------------
Title: Virgin Suicides, The (1999)
genres: Drama|Romance
------------------------------------------------------------------------
Title: Corpse Bride (2005)
genres: Animation|Comedy|Fantasy|Musical|Romance
```

CONCLUSIONS

Ant Colony Optimization and the Genetic Algorithm have proven to be a useful tool for developing a movie recommendation system. The highest rating accuracy score found was 0.83 with the following parameters (alpha =0.2, beta = 0.8, num movies =20, num agents = 30, num iterations= 50). While this might not seem like a very impressive score, it is important to remember that rating accuracy is calculated as a percentage of total possible stars on a 5-star system. So a rating of 0.83 * 5 = 4.15, this means that the average ratings of all overlapping recommendations in this iteration of the model is 4.15,which I believe is a great score.

A definite weakness of both the model and the testing method is the overlapping movies. The only way that we would know if a movie were a good recommendation for a test user would be if the prediction existed in the test set. As such, while we can gain a good understanding of how well the model can recommend movies that the user will enjoy, it does not measure how good the system is at recommending novel unseen movies. The only way to truly evaluate that would be with a live test audience.

Furthermore, our results when experimenting with different generations of users showed that the genetic algorithm was indeed an important addition that aided the system in making more accurate predictions. The system is also capable of adequate results when performing simple filters by genre.

NEXT STEPS/FUTURE IMPROVEMENTS

As mentioned previously, time did not permit the construction of a proper GUI for this movie recommendation system. As such, a natural extension of this project would be to build a GUI. This GUI would allow the user to explore the movies in an aesthetically and easy to navigate way. The addition of more complex query features can also expand the usefulness of the system. Given even more time one could create a user system that would keep track of a user's watched movies and periodically update the recommendation model.

Another natural extension of this project would be to compare it to other recommendation systems. While this paper makes a case for the efficacy or usefulness of ACO and GA for movie recommendations without comparisons to other recommendation systems we can not objectively rate its performance. The use of a live test audience would also be a great way to test the generalizability and "true" recommendation ability of this system.

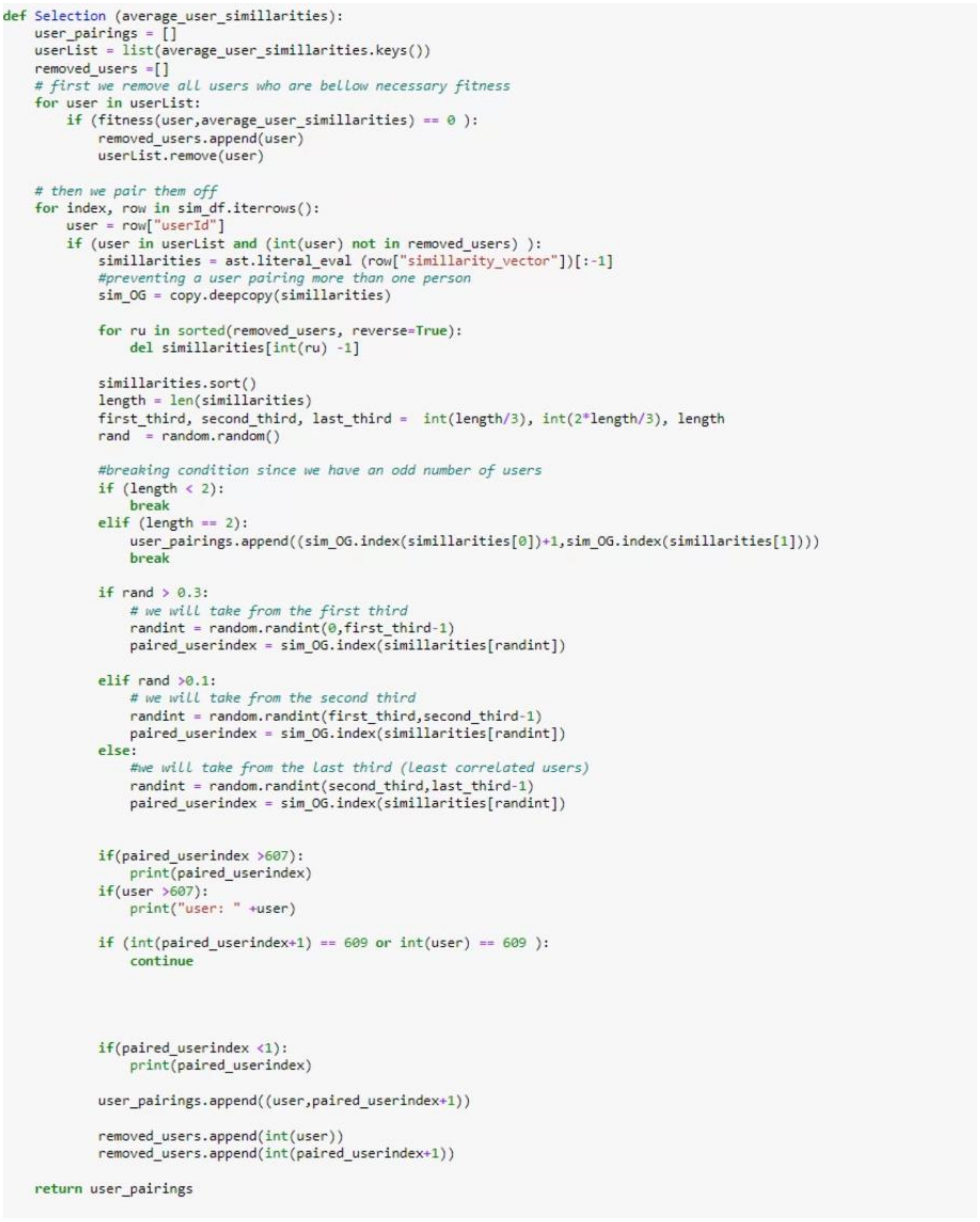
One improvement that should be made to the system is developing a more representative similarity function. This function is at the core of both the ACO and the GA component of this project, if the similarity metric were more representative of "true" Inter-user similarity then if would boost the overall performance of the entire system.

REFERENCES

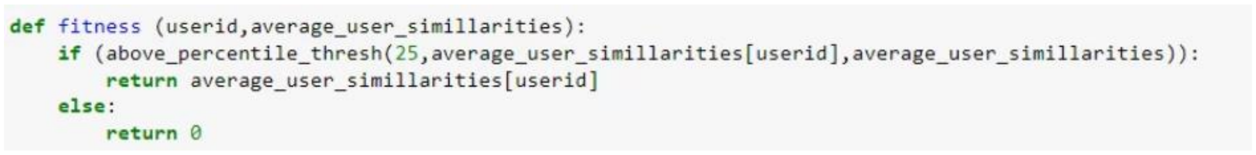
1. Barman, D. Sarkar, R. Tudu A. Chowdhury, N.(2020) Personalized query recommendation system : A genetic algorithm approach. Journal of Interdisciplinary Mathematics. Volume 23, Number 2, pp. 523–535
2. Dorigo, M. Briattari, M. Stutzle, T. (2006) Ant *colony Optimization*. EEE Computational Intelligence Magazine, volume 1, number 4, pp. 28-39. doi: 10.1109/MCI.2006.329691
3. Harper, F.M. Konstan, J.A. (2015) The MovieLens Datasets: History and Context. ACM Transactions on Interactive Intelligent Systems, Volume 5, Number 4, pp 19:1–19:19
4. Kim, H. Kim, P. (2017) Reliability–redundancy allocation problem considering optimal redundancy strategy using parallel genetic algorithm. Reliability Engineering & System Safety, Volume 159, pp. 153-160
5. Lingaraj, H. (2016). A Study on Genetic Algorithm and its Applications. International Journal of Computer Sciences and Engineering. Volume 4, pp. 139-143.
6. Manoharan, S. Senthilkumar, R. Jayakumar, S. (2022) Optimized multi-label convolutional neural network using modified genetic algorithm for popularity based personalized news recommendation system. Concurrency and Computation: Practice and Experience, Volume 34, Number 19, pp. na
7. Sethi, D. Singhal, A. (2017) *Comparative analysis of a recommender system based on ant colony optimization and artificial bee colony optimization algorithms*. 8th International Conference on Computing, Communication and Networking Technologies (ICCCNT). pp. 1-4. doi: 10.1109/ICCCNT.2017.8204106.
8. Sobecki, J. & Tomczak, J. (2010). Student Courses Recommendation Using Ant Colony Optimization.
   pp. 124-133. Doi: 10.1007/978-3-642-12101-2_14.

APPENDICIES

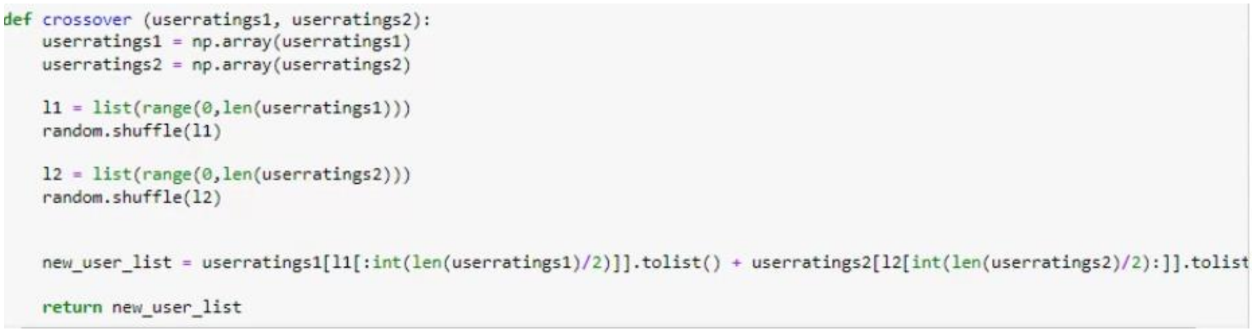
APPENDIX A: GENETIC ALGORITHM FUNCTIONS



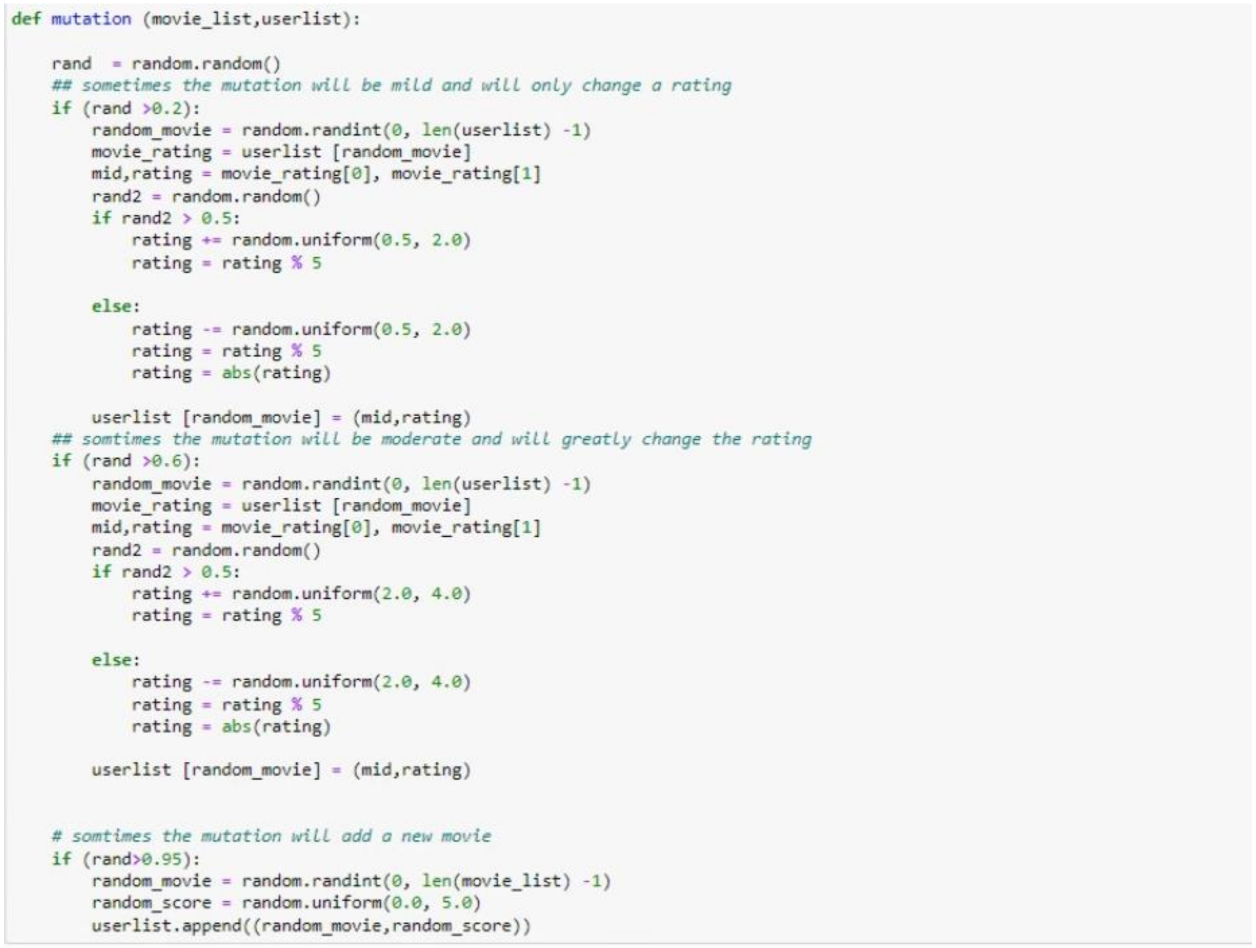
```
def Selection (average_user_simillarities):
    user_pairings = []
    userList = list(average_user_simillarities.keys())
    removed_users =[]
    # first we remove all users who are bellow necessary fitness
    for user in userList:
        if (fitness(user,average_user_simillarities) == 0 ):
            removed_users.append(user)
            userList.remove(user)

    # then we pair them off
    for index, row in sim_df.iterrows():
        user = row["userId"]
        if (user in userList and (int(user) not in removed_users) ):
            simillarities = ast.literal_eval (row["simillarity_vector"])[:-1]
            #preventing a user pairing more than one person
            sim_OG = copy.deepcopy(simillarities)

            for ru in sorted(removed_users, reverse=True):
                del simillarities[int(ru) -1]

            simillarities.sort()
            length = len(simillarities)
            first_third, second_third, last_third =  int(length/3), int(2*length/3), length
            rand  = random.random()

            #breaking condition since we have an odd number of users
            if (length < 2):
                break
            elif (length == 2):
                user_pairings.append((sim_OG.index(simillarities[0])+1,sim_OG.index(simillarities[1])))
                break

            if rand > 0.3:
                # we will take from the first third
                randint = random.randint(0,first_third-1)
                paired_userindex = sim_OG.index(simillarities[randint])

            elif rand >0.1:
                # we will take from the second third
                randint = random.randint(first_third,second_third-1)
                paired_userindex = sim_OG.index(simillarities[randint])
            else:
                #we will take from the last third (least correlated users)
                randint = random.randint(second_third,last_third-1)
                paired_userindex = sim_OG.index(simillarities[randint])


            if(paired_userindex >607):
                print(paired_userindex)
            if(user >607):
                print("user: " +user)

            if (int(paired_userindex+1) == 609 or int(user) == 609 ):
                continue



            if(paired_userindex <1):
                print(paired_userindex)

            user_pairings.append((user,paired_userindex+1))

            removed_users.append(int(user))
            removed_users.append(int(paired_userindex+1))

    return user_pairings
```



```python
def fitness (userid,average_user_simillarities):
    if (above_percentile_thresh(25,average_user_simillarities[userid],average_user_simillarities)):
        return average_user_simillarities[userid]
    else:
        return 0
```

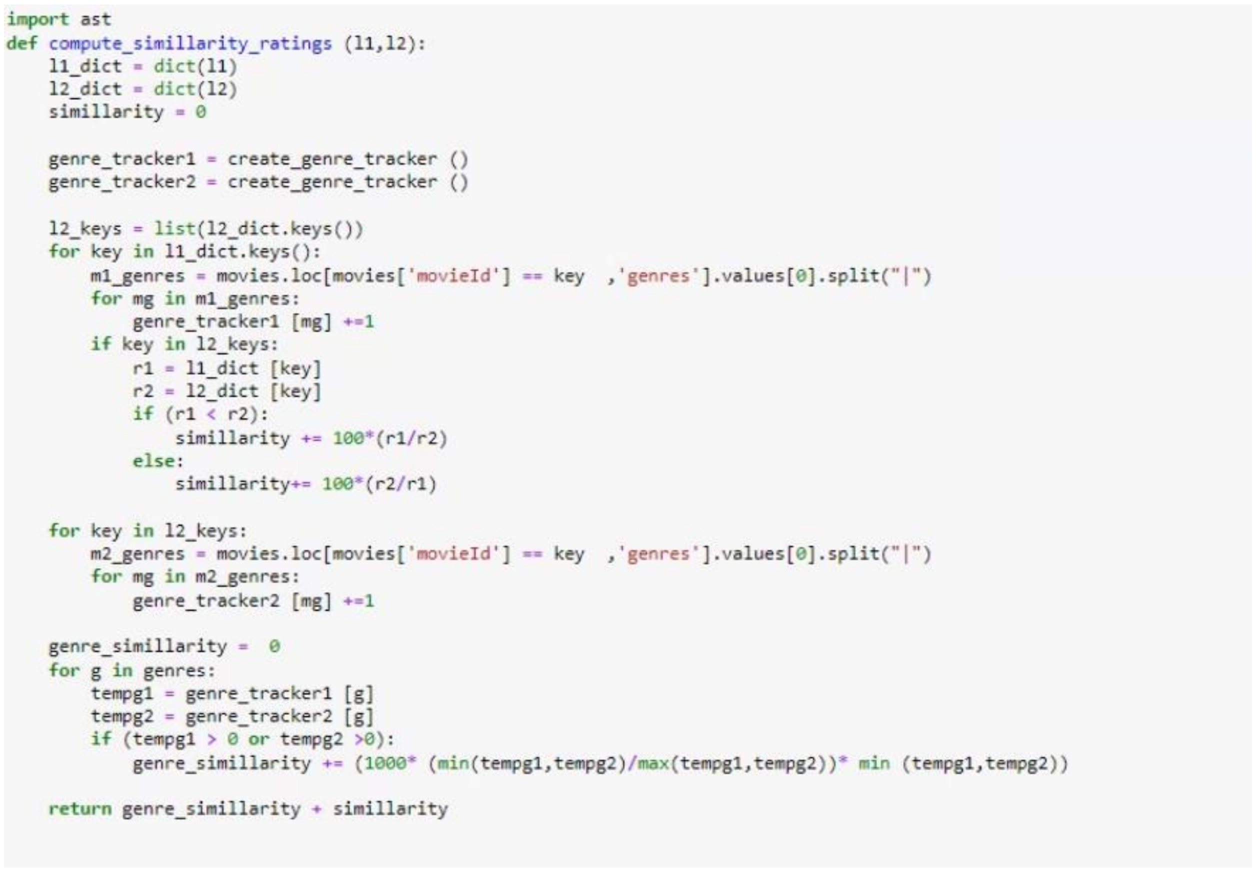


```python
def crossover (userratings1, userratings2):
    userratings1 = np.array(userratings1)
    userratings2 = np.array(userratings2)

    l1 = list(range(0,len(userratings1)))
    random.shuffle(l1)

    l2 = list(range(0,len(userratings2)))
    random.shuffle(l2)


    new_user_list = userratings1[l1[:int(len(userratings1)/2)]].tolist() + userratings2[l2[int(len(userratings2)/2):]].tolist

    return new_user_list
```



```python
def mutation (movie_list,userlist):

    rand  = random.random()
    ## sometimes the mutation will be mild and will only change a rating
    if (rand >0.2):
        random_movie = random.randint(0, len(userlist) -1)
        movie_rating = userlist [random_movie]
        mid,rating = movie_rating[0], movie_rating[1]
        rand2 = random.random()
        if rand2 > 0.5:
            rating += random.uniform(0.5, 2.0)
            rating = rating % 5

        else:
            rating -= random.uniform(0.5, 2.0)
            rating = rating % 5
            rating = abs(rating)

        userlist [random_movie] = (mid,rating)
    ## somtimes the mutation will be moderate and will greatly change the rating
    if (rand >0.6):
        random_movie = random.randint(0, len(userlist) -1)
        movie_rating = userlist [random_movie]
        mid,rating = movie_rating[0], movie_rating[1]
        rand2 = random.random()
        if rand2 > 0.5:
            rating += random.uniform(2.0, 4.0)
            rating = rating % 5

        else:
            rating -= random.uniform(2.0, 4.0)
            rating = rating % 5
            rating = abs(rating)

        userlist [random_movie] = (mid,rating)


    # somtimes the mutation will add a new movie
    if (rand>0.95):
        random_movie = random.randint(0, len(movie_list) -1)
        random_score = random.uniform(0.0, 5.0)
        userlist.append((random_movie,random_score))
```

APPENDIX B: SIMILLARITY FUNCTION



```
import ast
def compute_simillarity_ratings (l1,l2):
    l1_dict = dict(l1)
    l2_dict = dict(l2)
    simillarity = 0

    genre_tracker1 = create_genre_tracker ()
    genre_tracker2 = create_genre_tracker ()

    l2_keys = list(l2_dict.keys())
    for key in l1_dict.keys():
        m1_genres = movies.loc[movies['movieId'] == key  ,'genres'].values[0].split("|")
        for mg in m1_genres:
            genre_tracker1 [mg] +=1
        if key in l2_keys:
            r1 = l1_dict [key]
            r2 = l2_dict [key]
            if (r1 < r2):
                simillarity += 100*(r1/r2)
            else:
                simillarity+= 100*(r2/r1)

    for key in l2_keys:
        m2_genres = movies.loc[movies['movieId'] == key  ,'genres'].values[0].split("|")
        for mg in m2_genres:
            genre_tracker2 [mg] +=1

    genre_simillarity =  0
    for g in genres:
        tempg1 = genre_tracker1 [g]
        tempg2 = genre_tracker2 [g]
        if (tempg1 > 0 or tempg2 >0):
            genre_simillarity += (1000* (min(tempg1,tempg2)/max(tempg1,tempg2))* min (tempg1,tempg2))

    return genre_simillarity + simillarity
```