PageRank-Based Movie Recommendation System

Aayush Sabharwal 2019CSB1060 Harsh Priyadarshi 2019CSB1088 Gharmanshu Patanjali 2019EEB1159

November 2020

1 Abstract

Our project goal was to build a functional movie recommendation system using publicly available data. To this end, we used the GroupLens MovieLens 25M dataset, and built a web scraper to extract other information from IMDb. We cleaned, filtered and analysed our data to come up with a workable plan for constructing an adjacency matrix of movies. This is constructed by taking a linear combination of various similarity matrices made using different similarity parameters, such as director, tags, genre, etc. The coefficients of the linear combination were calculated by running gradient descent multiple times to minimize an error function. We used anonymized user ratings data to subset this matrix based on user input, run PageRank on the subset, and return the highest recommended movies. We also built a GUI for this application.

Drive Link(Included since we have several large matrix files that could not be uploaded to GitHub. This does not contain any intermediate matrices.)

Demo Video

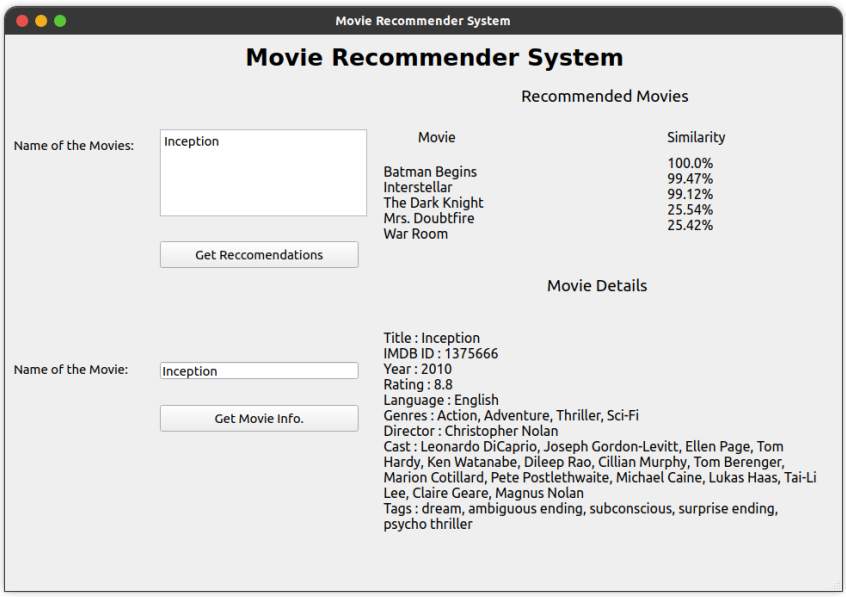

Fig: A screenshot of the recommendation system GUI

2 Introduction

Recommender systems are algorithms that attempts to predict user preferences. For example, recommender systems are used widely by services such as Spotify and Netflix to recommend songs and movies respectively to users, based on their past preferences in the respective domains. Recommender systems have grown significantly, and they are core parts of various online content providers to the extent that said content providers offer significant rewards for improvements to their algorithms. This can be seen in Netflix Prize.

PageRank is an algorithm used to rank importance of nodes in a network. It works on the concepts of random walks, wherein a fictional person traverses the graph at random, and keeps track of the frequency of all the nodes visited. This eventually converges, and the frequency values represent relative importance of the nodes in the graph.

2.1 Problem

The aim for this project was to build a PageRank based movie recommendation system from first principles. By defining a similarity parametric for all pairs of movies using publicly available data, and thus creating an adjacency matrix, we can find the most important movies in a network using PageRank. Methods of making the recommendation user-specific are discussed later.

2.2 Literature

Our main inspiration and source of information were interactive sessions of CS522 course held during this semester. From there we learnt about PageRank and various methods of computing it.

2.3 New Idea

After learning about PageRank, and its implementations, we attempted to use the power iteration method of calculating PageRank to attempt to build a movie recommendation system.

3 Method

3.1 Implementation Details

3.1.1 Data Collection and Processing

The data for this project is all from publicly available sources. Our first goal was getting a comprehensive set of movies. Initially, we tried getting IMDb data, but realised this is only an exceedingly small and scattered subset. Eventually, we found and decided to use the GroupLens MovieLens 25M dataset, present in the ml-25/ directory. This consists of a relatively small set of around 62k movies, but this was more than enough for our purposes. In addition to just movies, this dataset also included for each movie it's title, IMDb ID, year of release, genre(s), tags, and perhaps most importantly 25 million (hence the name) anonymised user ratings for the movies. For additional details about the specific data contained in this dataset, refer to ml-25m/README.txt.

For additional data about each movie, we built a web scraper that used the IMDB ID to go to the IMDB URL and extract the relevant information. Information such as Genre, Relevant Tags, Director, Cast, Language and Rating was scraped from the IMDB web- site for each movie. Requests was used to get the webpage in HTML format, then

BeautifulSoup4 was used to scrape the required data. The script for the scraper can be found in scraped-movies-data/scraper/web_scraper.py.

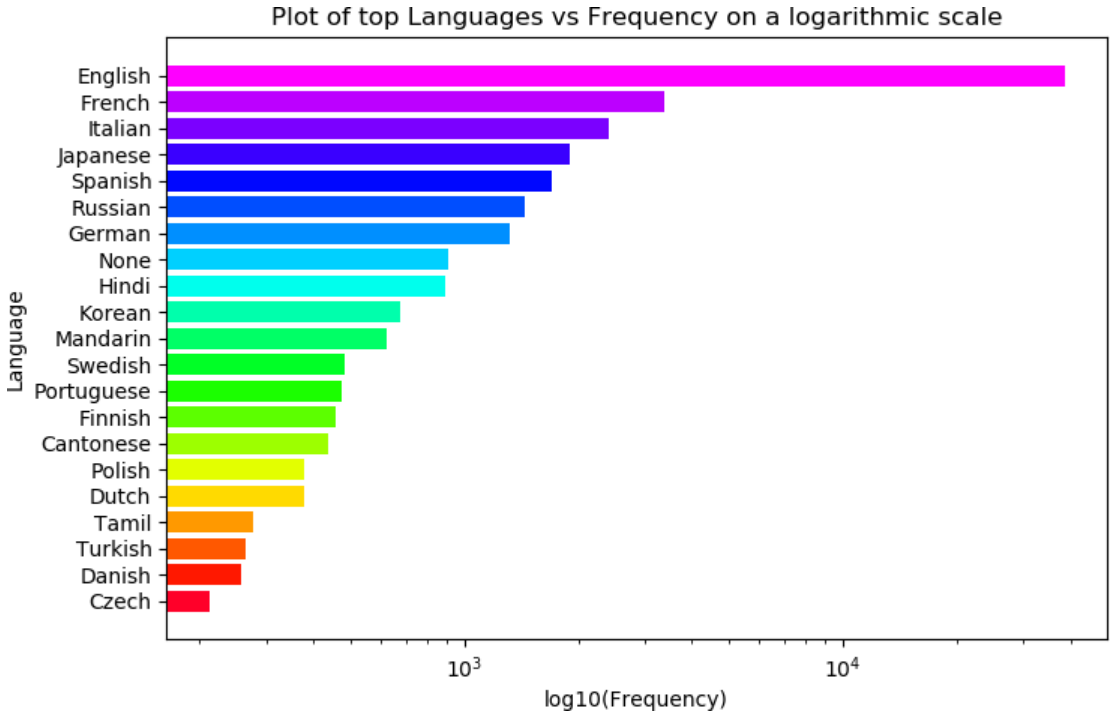

Fig: Bar chart displaying frequency of the most common languages in the dataset, on a logarithmic scale

To clean the movie data and collect it into a proper format, pandas and re libraries were used to handle the csv files and process text patterns respectively. This process is present in cleaned_movies_generator.py, and its output is present in ml-25m/clean_movies.csv. Additionally, most of the movies in the final dataset were irrelevant, primarily because they were of other languages. Additionally, we could not process the entirety of the data. For example, a single 62000 62000 matrix of half-precision (2 byte) floats that would represent similarity between movies would take up over 7GB of space. To this end, movies were filtered to include only those in English, Hindi, Urdu and Punjabi. Additionally, there were some movies whose IMDb IDs were invalid (the webpage resulted in a 404 error while scraping) so these movies were dropped. This resulted in a set of 23843 movies, which are available in processed_data/cleaned_subsetted_movies.csv.

Additionally, ml-25m/ratings.csv contained in addition to userID, movieID, and rating, the timestamp for each rating. Since this was of no use to us, the column was dropped, and the resulting dataset is in ml-25m/timeless-ratings.csv.

3.1.2 Methodologies Employed

Graph-centric Initially, we tried a graph-centric approach. In this method, the movies would be nodes on a weighted, undirected graph, and edges would indicate similar movies, the edge weight being the relative similarity for each movie. We chose graph-tool as our library of choice since in comparison to various other libraries, it came out significantly faster in various Benchmark tests.

The plan was to use various parameters of the collected data to create graphs that rep- resent those similarities. For example, using the user ratings a graph was created (graphs/ratings_graph.gt.xz) in which users and movies were linked by edges weighted with the rating given to that movie by that user. This was intended to be used to calculate one similarity parametric between two movies, by using the fact that similar movies would have fewer edges connecting them, and these edges would have higher weight. This allowed us to then create a parametric, where the similarity between two movies ($S_{u,v}$) is directly proportional to the number of shortest paths ($N_{u,v}$) (in terms of number of edges) between two movies, directly proportional to the average weight along the paths ($A_{u,v}$), and inversely proportional to the number of edges in the paths ($E_{u,v}$). Mathematically,

$$S_{u,v} = \frac{N_{u,v} A_{u,v}}{E_{u,v}}$$

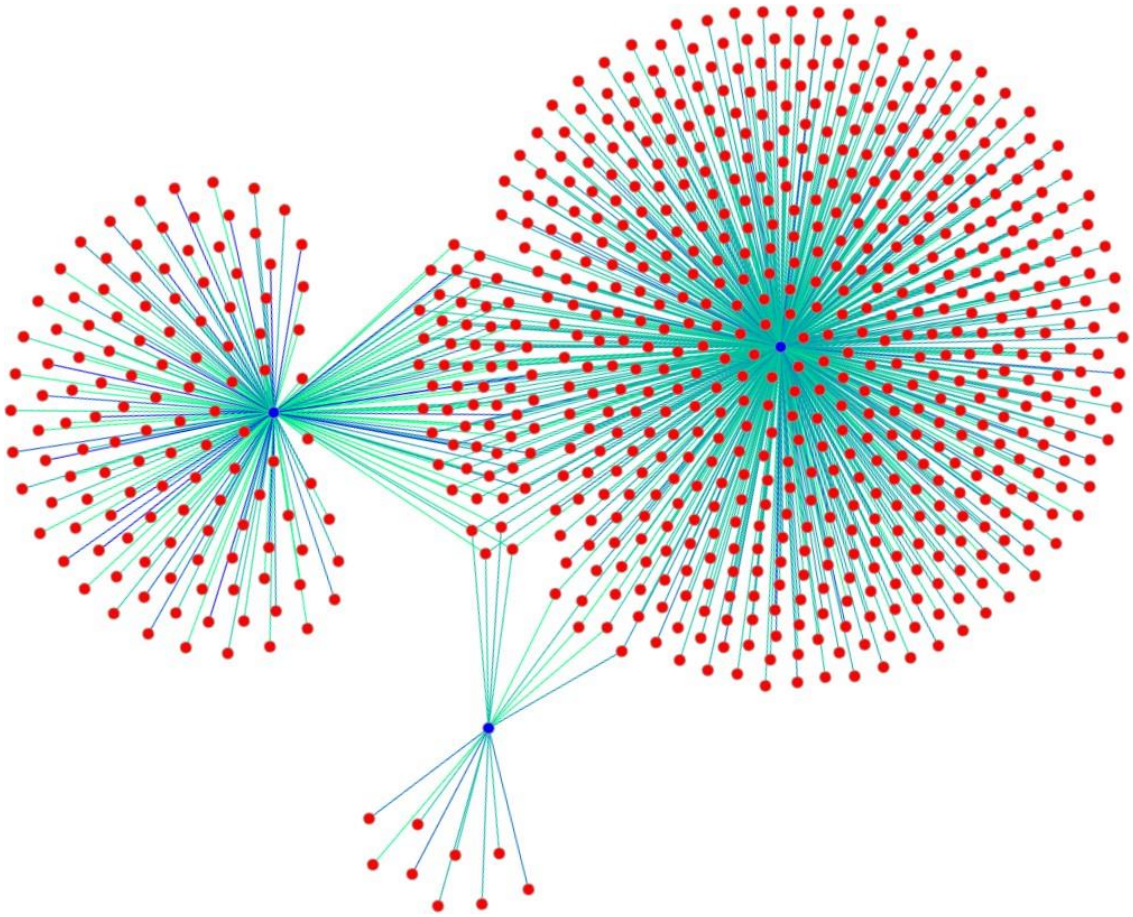

Fig: Graph of user ratings, where blue nodes at centers are users, red nodes are movies, and edges are weighted by the rating given by user to a movie (shown through colour, green edges indicate higher ratings). This only includes 3 users and the movies they rated.

This approach quickly fell apart when we constructed the graph, which turned out to be extremely slow to process, and impossible to run many centrality or relatedness algorithms on simply due to the large memory requirements.

Matrix-centric After the lecture on PageRank and calculating it using power iteration, we at- tempted to represent movie similarities as matrices. For this, we identified parameters using which we can compare movies. These are director, cast, tags, genre, user ratings and language. We attempted to use h5py library that allows for storing large matrices on disk as .hdf5 files. While this method would have worked in concept, the processing times were too long since reading from and writing to disk is an awfully slow process. It was at this point when we realised that 62000 is too large a set of movies to compute on, even with low-precision datatypes. As a result, we subsetted the movies as was described above.

Our goal at this point was to calculate one matrix for each similarity parameter and take a linear combination of these matrices to obtain the final adjacency matrix representing similarities between all pairs of movies. Director, genre, language and rating similarities were quick to calculate. In the case of genres, this was because there were very few total genres in the dataset. For director, language, and rating, the process of computing similarities was not computationally expensive.

Cast similarity and user similarity had the same issue, that the number of different users/cast members was too large to compute. Theoretically, our method would benefit from being able to compute the cast similarity as well, however due to restriction of our systems this was not possible. User similarity was incorporated in two separate ways. Firstly, the 20,000 users with the most ratings were used to construct a user similarity matrix, which was then used to compute an error function that enabled calculating coefficients of the linear combination through gradient descent. This method is explained in more detail in the next section. The second application is used to subset the movies to run pagerank on. This method is explained in a latersection.

All the similarity matrices had different ranges of values, depending on how they were calculated. Except for director and language similarities, the matrices also had the issue that larger values represented greater differences, instead of greater similarities. To rectify this, and normalize the values, 1 was added to each cell of the matrix, and then reciprocal of all values was taken. This normalises the values, since 0 cells along the diagonal and other places (indicating 100% similarity in that metric) become 1, and larger difference values are less than 1. This added a constraint on the linear combination coefficients, that they had to sum to 1 for the scale to remain the same. This is a workable restriction.

All the similarity matrices can be found in the all_similarities/ directory. The code to calculate rating similarity is in all_similarities/calculation/rating-similarity.py. For director similarity, it is in director-similarity.py in the same directory. The same proc. was applied for language similarity. Genre and tags similarity was calculated using the method in similarity-calculator.py. User similarity was calculated using user-similarity-calculator.py. Raw, testing code can be found in movie-similarity-generation.ipynb. The postprocessing code to add 1 to and calculate reciprocal of the matrices is also present there.

3.1.3 Calculation of Linear Combination Coefficients

To calculate the coefficients for the linear combination coefficients of the similarity matrices, gradient descent was employed. The user similarity matrix calculated previously was used as a target matrix. The error function was calculated as the sum of absolute difference between corresponding values of the adjacency matrix and the user matrix. Using scipy.optimize.minimize function and method='BFGS' gradient descent was run from multiple different starting values, and the best set of weights obtained was used for the final adjacency matrix. The code for this is available at weights-gradient-descent.py.

3.1.4 Final Recommendation System

After the adjacency matrix was obtained from the coefficients calculated through gradient de- scent, PageRank was implemented using the power iteration method. The input movie names are matched to the ones in our dataset through fuzzy matching, using fuzzywuzzy library for Python. This operates by calculating the Levenshtein Distance for each title in our database, given an input title, and picks the one with the highest value.

After matching movie titles, these are mapped to their respective movie Id values. This is given as input to user_id.subset() which subsets around 3000 movies on which the pagerank algorithm is supposed to be run. To make sure that the recommender system is user-specific, the movies are subsetted based on the movies that are given as input by the user. Genres of the movies given in the input is used to subset the users. A user-genre matrix is created, each cell showed the weights (proportional to the rating they gave to a movie of that genre) of a particular genre as rated by a particular user. This is then normalized as there are extremes to the number of movies that a user rated. Final matrix can be found in processed_data/Scores_Normalized.csv. Users who rated less than 50 movies were removed from this matrix; they were supposed to be used as test set for gradient descent but later discarded as this data was not sufficient for calculating the error in similarity matrix. The users who are top rated for the weighted average of the genres that is provided as input is returned such that the number of movies rated by those users is close to 3000. The script for subsetting the movie can be found in user_id.py.

A collection is then formed comprising of all the movies rated by all the given users, along with the input movies. These movie Id values are given to the pagerank function. The function subsets the adjacency matrix, accordingly, makes it Markovian, and performs PageRank using the power iteration method. To improve locality of results, a reset probability is introduced. Each column corresponding to the movies given by the user is given an additional 0.15 weightage, by the following formula

$$M^{J} = (1 - a)M + aR$$

Where M^{J} is the final matrix on which PageRank is performed, M is the raw subsetted matrix, R is a matrix of the same dimensions as M with columns corresponding to input movies as 1, and a = 0.15 is the reset probability

The starting vector for power iteration is taken as v = (1, 0, 0, ..., 0), and convergence is achieved than tolerance $G = 10^{-4}$. Mathematically, convergence occurs if when the maximum difference between corresponding values in successive iterations of v is less

$$max(v_i - v_{i-1}) \leq G$$

Where v_i is the value of the vector v after the ith iteration. A cap of 1000 on iterations was implemented as well, to prevent an infinite loop in the rare case that convergence does not occur.

The function returns a dictionary mapping movieId to PageRank, of which the 5 largest values are the recommended movies for the given input.

3.1.5 Tools and Libraries Used

All the code for this project is written in Python 3.8.5. The libraries used in this project and their usages are as follows

(Last column denotes whether library is required to run the recommender system through gui.py)

Library	Usage	Link	Required
numpy==1.19.4	Various mathematical applications, including handling of very large matrices, matrix multiplication, and various other matrix operations	https://numpy.org/	Yes
matplotlib==3.1.2	For data visualizations	https://matplotlib.org/	No
scipy==1.3.3	Used scipy.optimize to perform gradient descent and find optimal combination of weights for different similarity parameters	https://www.scipy.org/	No
pandas==1.1.3	For handling and processing of raw and processed data, in the form of .csv files	https://pandas.pydata.org/	Yes
re	Python Regular Expressions library for pattern matching while processing raw data	https://docs.python.org/3/library/re.html	No
numba==0.51.2	A Python compiler that allows for compiling a subset of python and numpy to machine code, used while testing methods and processing data to accelerate some computations	https://numba.pydata.org/	No

Library	Usage	Link	Required
graph-tool	A Python library written in C++ for handling graphs, used while testing different methods of PageRank and methods of processing movie similarity. This was eventually not used in the result, and is not required to run the recommender system	https://graph-tool.skewed.de/	No
lxml	A Python library that aids in processing large XML and HTML files	https://lxml.de/	No
BeautifulSoup==4.9.0	Beautiful Soup is a library that helps to scrape information from web pages, used while scraping the IMDB site for data.	https://pypi.org/project/beautifulsoup4/	No
fuzzywuzzy==0.18.0	Used for fuzzy matching of input movie names to those in our dataset	https://pypi.org/project/fuzzywuzzy/	Yes
python-Levenshtein	Used by fuzzywuzzy to accelerate calculation of Levenshtein distance	https://github.com/ztane/python-Levenshtein/	Yes
PyQt5	PyQt is a Python binding of the cross-platform GUI toolkit Qt, implemented as a Python plug-in, used for making a GUI to run the program	https://pypi.org/project/PyQt5/	Yes

4 Results

4.1 Experiment Findings

The final adjacency matrix obtained is final_adj_mat.npz. The weights used to obtain this were (0.873, 0.047, 0.08) for director, genre, and tags similarity, respectively. Including additional matrices took a long time to compute, and for the time we were able to run, the error was higher.

Additionally. It was found that Drama and Comedy movies seem to dominate the recommendations. The recommendation system can be interacted through the GUI app, by running `gui.py`.

4.2 Interpretation of Findings

Observing the weight values, it appears as though director similarity is given a surprisingly high weightage. This may make some sense since directors often make similar types of movies. Genres was also surprisingly given a fairly low weightage. This could possibly be because tags serve a similar purpose, and since there are a larger number of tags, the granularity of similarity values would be much more, and consequently less polarising.

Upon exploration, it appeared that Drama and Comedy were very frequent genres in our dataset. Due to this, while using `user-id.subset` they have disproportionately higher influence on the subsetted users. This tends to bias the recommendations towards movies with these genres, since they are the ones subsetted. Currently, movies without these tags have better recommendations.

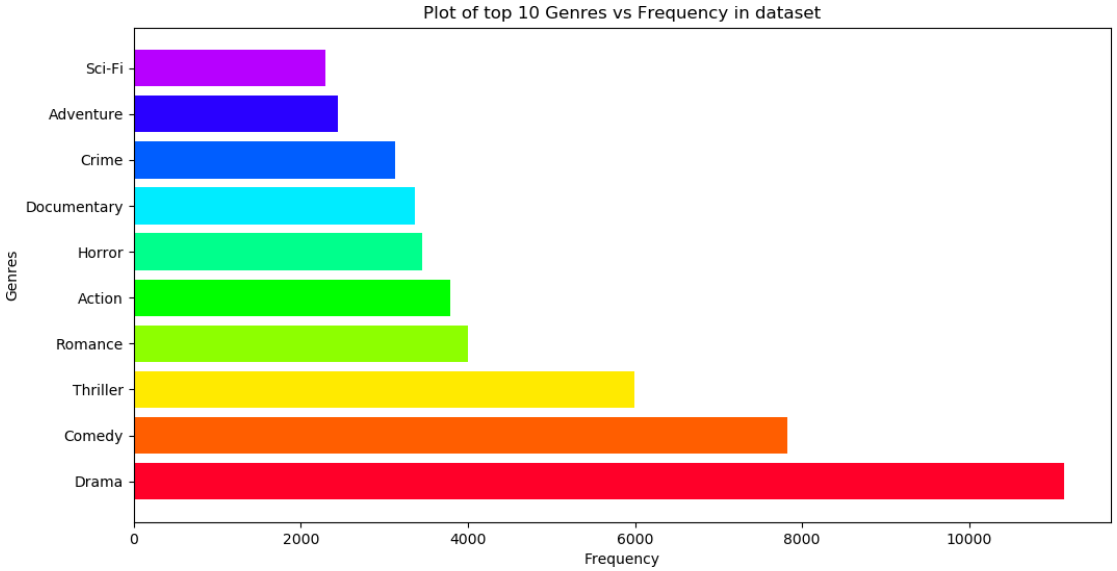

Fig: Bar chart displaying frequency of the top 10 genres in the dataset we used, on a linear scale. Visibly, Comedy and Drama genres are exceptionally frequent

5 Conclusion

PageRank appears to be a workable technique for building a recommendation system. It may be possible to improve the system further if larger data can be processed, but this project can serve as a proof of concept for such an approach. There are definitely various improvements that can be made to the approaches we have taken to process and use our data, but the current approach

has also provided us with workable results. Some improvements could be in the form of finding an appropriate normalization for different frequencies of genres, to avoid biasing. Additionally, the algorithm could be significantly improved if additional data and the other matrices are able to be processed and used in the similarity metric. We did not have a deep understanding of the various methods of gradient descent, and perhaps a better method, implemented appropriately, could have offered better results. In relation to this, there could be significant improvements to the error function used for gradient descent.

This project was a major learning experience, in various aspects of data analysis and recommendation systems. We learnt to scrape web data, process data with pandas and numpy, storing and manipulating large data, build GUIs, and applications of scientific techniques like gradient de- scent.

5.1 Team Work

This project was definitely a huge group effort. We managed to effectively split the workload by distributing different tasks and combining the results. Another method of collaboration was splitting the computation. For time consuming processes, like scraping web data, we were able to effectively halve the total time by each doing half the computation. This was applied not only for scraping web data, but also calculating tags and user similarity matrices, and running gradient descent for various input weights to find the best combination of coefficients.

Not only just computation and effort, but ideas are also much better with groups. We were able to effectively find faults in each other's ideas, to identify areas of improvement, and effective methods of building a better recommendation system. The project involved not only a lot of coding, but a lot of discussions held online to find solutions to various problems encountered. We also spent substantial amounts of time exploring and getting to understand the data, then sharing our findings to think of better pipelines to process our data.