A Social Media-Based Movie Recommendation System

Vinayak Ahluwalia[1], Harry Wang[1], Anthony Liang[1], Shaeq Ahmed[1], Joshua Israel[1]

[1]Department of Electrical Engineering and Computer Science, University of Michigan–Ann Arbor

Abstract

Due to the popularity of movies today, people would like personalized recommendations of new movies to explore. However, we have not found other researchers who have utilized social media, which has a wealth of information about an individual's preferences, to make recommendations. We present an individualized movie recommender system that crawls a user's various social media profiles, downloads and preprocesses the text, and then performs a cosine similarity measurement to retrieve a ranked list of ten recommended films. The user then provides feedback on the films, facilitating a query reformulation and the retrieval of more relevant films. We found increased Kendall-Tau, mean average precision, and mean reciprocal rank scores after query reformulation and overall upward trends in IR evaluation metrics due to collaborative filtering.

1 Problem Statement

With streaming giants like Netflix and Hulu garnering thousands of users to accompany a vast streaming database, users want to have personalized recommendations without having to look for them. At the same time, social media profiles provide a plethora of information about any single user's preferences, and can be used to generate recommendations for movies. As a result, text-based video retrieval methods can be used, which involve searching for a certain video using text metadata, which in this case refers to movie scripts.

We present a cosine similarity and collaborative filtering based algorithm that can crawl social media profiles and return recommendations with the highest similarity. Moreover, users would also like more effective ways to give feedback to recommender systems, which is why we experiment with different query reformulation methods.

2 Related Works

Berkovsky, S. and Freyne, J. (2015) and Huang, Y., Contractor, N. and Yao, Y. (2008) gave us insight into extracting information about users from social media. They provided background into how different attributes of a user can be inferred from his/her social media information. The references also gave insight into how these attributes are then used to recommend products to users. In the case of our project, the products are the movies recommended to each user. Discussions on how different attributes should be weighted was also included. This gave our group the idea of converting social media and movie scripts into feature vectors with TF-IDF. The TF-IDF structure allowed our framework to have a built-in way to capture multiple features and give these features different weights depending on their importance.

Zhang, F., Yuan, N., Lian, D., Xie, X. and Ma, W. (2016) and Rashid, A., Karypis, G. and Riedl, J. (n.d.) talked about how to use feedback and previous to improve recommendations. This inspired the group to incorporate query formulation as a means of leveraging user evaluation. The references also discussed using collaborative filtering as a way to gain insight on users that have never previously used a system before. There was also specific mention of using K-Nearest Neighbor (KNN) to assist in recommendations. This inspired us to use KNN to appropriately increase and decrease

the weights of previously relevant and irrelevant movies of the previous most similar user. This mechanism, in conjunction with our query reformulation, and TF-IDF weighting scheme which was used for ranking, contributed to the novelty of our project. Our project comprehensively synthesized and integrated these concepts into a recommender system that effectively leverages social media specifically for recommending movies. The application of these principles in this specific domain is something we found to be never or not commonly done before.

Seo, J. (2018) and Koehrsen, W. (2018) also gave us insights into designing our recommender system. However, due to time constraints, we were unable to implement the ideas in these references. Koehrsen, W. (2018) discusses using neural networks to assist in recommendations. However, due to our limited data (which needed to be collected from real users that we knew), we found training a neural network to be impractical. Seo, J. (2018) explores using matrix factorization to fill in missing data, such as project scores given a certain user-movie pair. This would be a great future direction to explore in order to improve our collaborative filtering mechanism. Overall, we were able to synthesize many different concepts from these resources to create a comprehensive system that uniquely leverages the principles discussed in a targeted fashion.

3 Graphical User Interface

Flask was used to create the application, integrating the frontend, coded in HTML and CSS, with the backend in Python. The GUI runs as a web application. The home page has four fields, two which take in the Facebook login and password, and two other fields with links to the user's Facebook and an optional Twitter link.

Once the user hits "Get Recommendations," the system will run and the ten movies will be returned, as shown in Figure 1. Each movie will have a field where the user will enter his/her ranking and select a button labeled "Relevant" or "Not Relevant." Afterwards, the user will select one of the three relevance feedback methods and hit "Submit Feedback," which will run query reformulation and return a new list of movies.

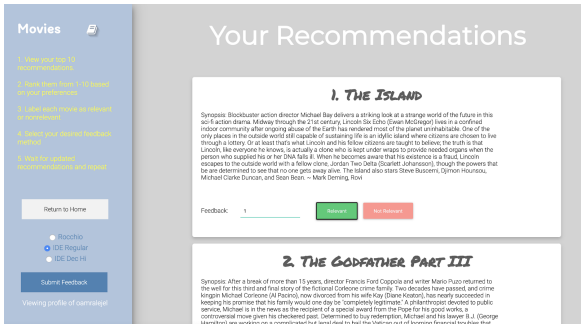

Figure 1: Graphical User Interface for Prototypical User's Recommendations

4 Social Media Scrapers and Datasets

We created a scraper using the Selenium python framework to scrape a given user's social media profiles. Selenium allows us to automate web browsers functionalities such as clicking, scrolling, typing, and retrieving HTML DOM elements. Currently we have scrapers for Facebook and Twitter. After a user enters their profile links and sends the request to generate recommendations, it will trigger the scraper. The scrapper will open the web browser, navigate to the specified profile, and start scrolling through it. We concatenate all the user's post into one text file which will serve as our "query" vector. This query vector has a lot of extraneous information which we will filter in the preprocessing stage. Since we wanted to make this into a web application that runs locally, it is important for us to automate the scraping process.

The other datasets used included 875 opens-source movie scripts from imsdb.com, a dataset from Kaggle that contains tags for a set of movies, and synopses for certain movies from rottentomatoes.com. All these datasets helped contribute to constructing the inverted index and the TF-IDF vectors for the

movie documents. However, there were some issues during data collection. For one, the Kaggle dataset also had tags relating to cast members and other movie personnel, which we originally hoped would match with celebrities that users followed on social media, and would give better movie recommendations. Unfortunately, we were unable to parse the cast and personnel names from the dataset due to the specific structure of the dataset. Moreover, we were unable to retrieve a synopsis for each movie from rottentomatoes.com because we used a scraper that converted each movie title into a URL, and sometimes the constructed URL would lead to a non-existent page. Lastly, initially we wished to retrieve a link to a movie poster for each movie, but we were unable to find a way to do so.

5 Vector-Space Model Approach

The basis of the system revolved around the ability to represent each movie and each user's social media profile using the vector space model. Each movie and each social media profile would be represented as a TF-IDF vector, and then the cosine similarity score between the social media profile, serving as the query, and each movie, serving as the document, would be calculated. Eventually, the ten movies with the highest cosine similarity scores would be returned to the user.

Before the TF-IDF vectors could be calculated, an inverted index was first formed, which maps each word in the vocabulary of a corpus to the documents in which it appears. In this case, the corpus was composed of all words from the 875 movies downloaded from imsdb.com, scraped synopses from rottentomatoes.com, as well as tags that could be retrieved for movies in the Kaggle dataset. It is important to note that text preprocessing was done before creating the inverted index, which involved tokenization using NLTK, stopword removal, and stemming using *porter_stemmer*.py.

The generation of the inverted index can be viewed in the function **create_data()** in *inverted_index.py*. Essentially, each of the 875 movie scripts was opened and all its text preprocessed and added to the inverted index. The title of the movie was processed to remove unnecessary punctuation and it was arbitrarily assigned a movie ID (between 1 and 875) to more easily keep track of each item. A dictionary mapping each movie ID to the actual movie title can be found in *index_to_movies* and was written to a pickle file to ensure consistency between trials.

Then, if data existed from the Kaggle dataset regarding the genre of that movie, the tags were also added to the inverted index. Lastly, we attempted to retrieve the synopsis of each movie in the dataset from rottentomatoes.com by appending the movie's name to the URL, and if a synopsis was found, it's text was also added to the inverted index. The data structure for the inverted index is stored as *inverted_index* in the code.

However, the synopses served another purpose. In the recommendation system, not only were movie titles returned, but a synopsis, if available, was also returned to help the user determine the relevance of the recommended movie. This was important if the user had never the seen the recommended movie. A data structure called *synopsis* was created in *inverted_index.py* to map each movie ID to a tuple containing the synopsis (as a string) as well as a link to the movie poster. We were, however, unable to find links to the movie posters, so that feature went unused.

After the inverted index was complete the entire data structure was written to a pickle file, so that is could be quickly read in during execution of the application, instead of having to be reconstructed each time. Moreover, the inverted index helped construct the data structure known as *doc_term_weightings*, which is dictionary that maps each movie ID to an object containing its TF-IDF vector and the length

of the vector. This data structure was also written to a pickle file. The TF-IDF weight for each vector element was computed using the equation below:

Lastly, the application must actually be run. Once the user enters in the necessary links to their social media and the profile is scraped, the query is presented as a text document and undergoes the same preprocessing the movie documents underwent. However, special care was taken to remove first names and last names using the *names-dataset* library, remove non-English words using the *synsets()* function in NLTK, and finally remove words commonly appearing in social media that have no bearing on the user's preferences such as "happy" and "birthday." Afterwards, the query was converted into a TF-IDF vector and the cosine similarity value was computed between the query and each movie document vector in the database using the formula below.

In our search of related works, we did not come across researchers who had implemented a movie recommendation system utilizing cosine similarity between movie scripts and social media profiles. Moreover, we augmented our system utilizing collaborative filtering to improve recommendations for users using past data, which will be discussed in the next section.

6 Collaborative Filtering

For Collaborative Filtering, we implemented a K-Nearest Neighbor Algorithm (KNN). Everytime a user inputs their social media profile, his/her information is parsed into a query vector. KNN will then find the most similar previous user by finding the euclidean distance of the current query vector and each of the previous query vectors. The previous query with the smallest euclidean distance from the current query represents the most similar previous user. Each previous query vector is stored with a list of the corresponding relevant movies and a list of the correspond irrelevant movies. These lists represent the movies that were rated as relevant and irrelevant from the user that generated the previous query vector. Using this information, the similarity scores of the previously relevant and irrelevant movies are multiplied by certain factors for the current user. Previously relevant movies are multiplied by an upvoting factor, which should be above 1.0. Previously irrelevant movies are multiplied by a downvoting factor, which should be below 1.0. Our group tested several different upvoting and downvoting factors to see which ones produced the best results in terms of the quality of recommendations. We found that an upvoting factor of 1.05 and a downvoting factor of 0.95 to work the best from the experiments. While collaborative filtering is a common method that is applied in recommender systems, we found our mechanism of utilizing this concept to be unique in the sense that it adjusts the similarity scores, and thus the rankings of the document vectors (which used TF-IDF).

7 Query Reformulation

Query reformulation was essential to the system because only the user can gauge the relevance of the recommendations, and due to the relatively short time period it takes to generate recommendations, the user would hopefully receive better recommendations after engaging in more rounds of relevance feedback. The goal was that after a certain number of iterations of relevance feedback, the user would be presented with a list of all relevant movies in an ideal rank order.

The three methods that were explored were Rocchio, IDE Regular, and IDE Dec Hi. Originally, the optimal query reformulation was implemented, but we realized that the method only held theoretical, not practical value, so it was removed.

8 IR Evaluation Metrics

Three main IR evaluation metrics were used to test the movie recommender system:

Kendall-Tau, Mean Average Precision (MAP), and Mean Reciprocal Rank (MRR). All metrics relied on user feedback at the time of recommendation.

Kendall-Tau was used to compare the distances between the system-generated ranking of ten movies and the user-generated rankings of the same ten movies during relevance feedback. Kendall-Tau was implemented by generating all possible pairs of ranks between two movies for the system-generated rankings, and then counting the number of agreements between those pairs and the user-generated pairs.

Recall could not be calculated for the system because it is not possible to ascertain how many of the 875 possible movies are relevant for any particular user. As a result, we relied on metrics that could be calculated just using user feedback for the ten returned movies. MAP was used to gauge the number of relevant movies that the system returned and was calculated as the average of the average precision metrics for each user. Also, MRR tested whether the top recommendations were relevant or not, and was similarly calculated as the average of the reciprocal rank for all users.

The table below summarizes the resulting three metrics at the conclusion of testing. The three metrics represent the Kendall-Tau, AP, and RR that the user gave post-relevance feedback. This means that after the first round of feedback, the reformulated query returned an improved list of recommendations, and the user feedback on those recommendations constitutes the below data.

Evaluation Metric	Average Score
Kendall Tau	0.174
Mean Average Precision	0.790
Mean Reciprocal Rank	0.890

Table 1: Post-Relevance Feedback IR Evaluation Metrics for the System

Lastly, although it was not utilized, we created a functionality that wrote each Kendall-Tau, AP, and RR value to respective pickle files so the user would not have to manually keep track of the values.

9 Relevance Feedback

Besides the three numerical metrics discussed previously, we also wanted to determine which of the three relevance feedback metrics was best for the system, and to what degree they improved recommendations for any one user. To do this, a single user engaged in relevance feedback five separate times utilizing each of the three relevance feedback methods. The trends for Kendall-Tau and and MAP are shown in figures 2a) and 2b).

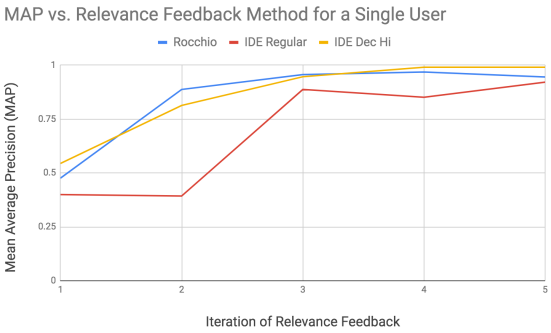

Figure 2a: All three relevance feedback metrics had similar upward trends in MAP for a single user after five iterations of relevance feedback

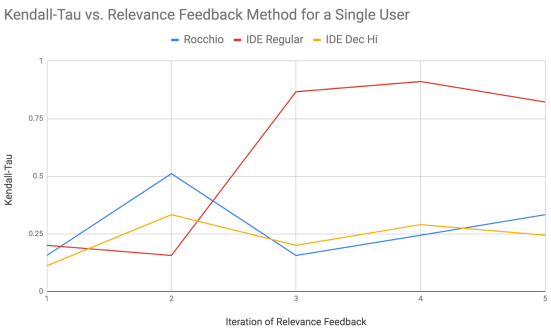

Figure 2b: IDE Regular Showed the Greatest Increase in Kendall-Tau Score for a Single User

The results shown in figures 2a) and 2b) indicate that while all three relevance feedback methods showed similar trends in MAP, meaning that all eventually resulted in the system returning 9 or 10 relevant movies,

the IDE Regular method ranked the returned movies in an order that best agreed with the user's feedback. As a result, IDE Regular was chosen to be the relevance feedback method of choice for testing and general application use. Such a decision paid dividends, as illustrated in figure 3, demonstrating an overwhelmingly positive increase in Kendall-Tau, AP, and RR for most users after relevance feedback and query reformulation. This demonstrates that the relevance feedback method is actually returning more relevant movies after user feedback and is generally ordering them such that it agrees more with user preferences.

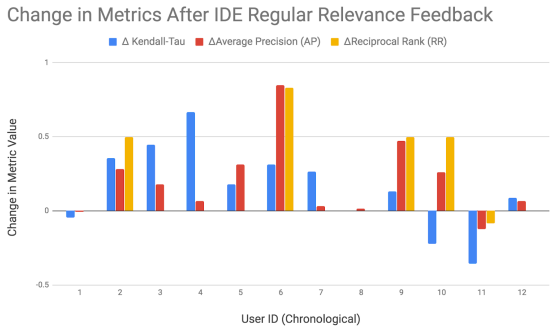

Figure 3: There was a general positive increase in Kendall-Tau, AP, and RR for recommendations given after relevance feedback and query reformulation

That being said, in figure 3, users 1, 8, and 11 showed either no improvement or even worsened performance with relevance feedback, which indicates that the IDE Regular is not a perfect method for improving recommendations. One thing that we would consider modifying in the future would be using the user-generated ranking of the movies to weight relevant movies such that during query reformulation, a relevant movie rated #1 or #2 by the user would have its document TF-IDF weights upweighted.

Lastly, we tracked the MRR and MAP of the system for pre-relevance feedback and post-relevance feedback as more users tested the system. The results are shown below in figure 4.

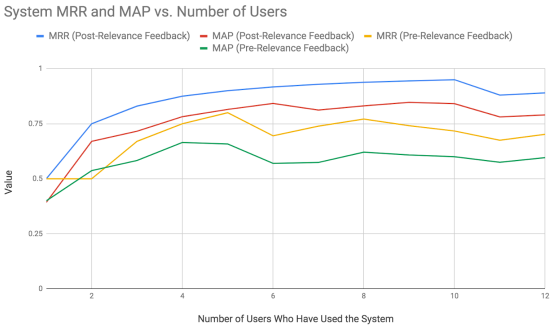

Figure 4: MRR and MAP generally trended upwards as more users tested the system, but eventually plateaued

Figure 4 demonstrates that MRR and MAP, both for pre-relevance feedback (first round of recommendations before query reformulation) and post-relevance feedback (second round of recommendations after query reformulation), generally trended upwards as more users tested the system but eventually plateaued after a certain number of users. Also, the MRR and MAP corresponding to pre-relevance feedback were consistently lower than the MRR and MAP for post-relevance feedback. This shows that the relevance feedback was a useful to the user for generating improved recommendations.

Moreover, the MRR at the conclusion of testing reached 0.89, and the MAP reached 0.79, which indicate the success of the system overall, even though much can be done to improve it. As mentioned before, the pre-relevance feedback MRR and MAP seemed to hold steady after eight users tested the system, whereas the MRR and MAP for post-relevance feedback still climbed until the tenth user. Since MRR and MAP are cumulative measures, these trends also exemplify the success of collaborative filtering, by which the system utilized past user feedback to improve recommendations for future users. As a result, the system began to learn to make better recommendations over time.

10 Discussion of Results

Although the system demonstrated considerable success, there are many improvements that need to be made. First, the system is quite inefficient; in Python, it takes on average 45-50 seconds to compute cosine similarity scores for a user, not accounting for the time to scrape and preprocess a social media profile. In order for this system to be more practically useful, it must be faster. We considered utilizing just the synopsis and Kaggle tags to create the inverted index, but ultimately decided that the entire movie script may include more relevant keywords, and provide better recommendations at the expense of increased computation.

Similarly, utilizing a movie script on its own is not a perfect indicator of a movie's relevance to a user. Many other factors go into this, such as production quality, critical reviews, cast members, acting quality, and era of production. For example, even though a user's profile may indicate that he or she enjoys superhero movies, for the system to recommend *X-Men Origins: Wolverine* may indicate an error on the system's part because even though the movie relates to superheroes, it is generally regarded as having a poor plot and bad acting. To compensate for this, in the future we would consider weighting cosine similarity scores by the scores on Rotten Tomatoes or figuring out a way to link each movie ID to cast members to account for users' preferences for actors.

In addition, the collaborative filtering, while it did show promise, may have actually hindered system performance towards the end of testing. We noticed that while testing, certain movies began appearing more and more often for different users, even though they had vastly different social media profiles. This indicated that certain movies being voted relevant by previous users drastically increased the chance that they would attain a high cosine similarity score for new users, and then actually be voted non-relevant. Future work could include experimenting with the weights of collaborative filtering to prevent this from happening.

11 Conclusion

Our movie recommendation system is the first to incorporate a similarity measure between social media profiles and movie scripts. In this sense, it's a text-based form of video retrieval using text metadata. While the system can take up to 60 seconds to find a list of ten recommended movies, the returned list of movies received generally positive feedback from users in the forms of Kendall-Tau, MAP, and MRR. Query reformulation resulted in even better recommendations, and IDE Regular appeared to be the best method for the system. Collaborative filtering would sometimes cause the same movie to appear for various users, but overall helped the system learn to give better recommendations.

Future work should include more sophisticated methods for machine learning such as grid-searching for relevance feedback and collaborative filtering. Also, a bigger movie dataset should be used to include more variety in movies. There should also be considerations to weight movie TF-IDF vectors by measures of critical ratings to upweight more classically "good" movies. Lastly, future work should also include methods to extract celebrities or topics of interest on the user's profile; for example, if the user follows Ryan Reynolds, then we should upweight movies with Ryan Reynolds.

12 Individual Contributions

Everyone worked together on the checkpoints, poster, and final report.

Anthony worked with Shaeq to automate the task of crawling Facebook and Twitter feeds, as well as designing the user interface. He also helped integrate the backend with the frontend. He also helped organize the poster's layout to include relevant information.

Harry wrote the entirety of *collaborative_filtering.py*, including parameter tuning for the collaborative filtering. He also worked with Vinny to code **rocchio()**, **ide_dec_hi()**, **ide_regular()**, **reciprocal_rank()** and **average_precision()** in *relevance_feedback.py*. He also parsed the tags and genre data from the Kaggle dataset which helped create the inverted index, and helped to extract useful information from the related works.

Joshua mainly coded within the file *relevance_feedback.py*. He helped code **rocchio()** and also coded **optimal_query()**, which was a function that was never used. He worked alone to code **kendall_tau()** and **sum_vector()**, and helped maintain style within the entire codebase.

Shaeq worked with Anthony to create the Selenium/Python scrapers that download text data from Facebook and Twitter from a user profile. Shaeq also created the frontend using Jinja templates and style sheets he created himself (custom CSS). He also worked on integrating the backend and frontend and worked on JS form validation for the recommendations and feedback page.

Vinny was mainly involved with the creation of the inverted index by downloading all the movie scripts, preprocessing them, creating the inverted index and document term weightings, and computing cosine similarity. As a result, he coded all the functions within *inverted_index.py* and wrote the pickle files containing the different dictionaries. He also worked with Harry to code the functions **submit_feedback()**, **reciprocal_rank()**, **average_precision()**, **ide_dec_hi()**, and **ide_regular()** in *relevance_feedback.py*. He also was responsible for conducting testing on different users, recording testing data, and creating the graphs.

13 References

1. Berkovsky, S. and Freyne, J. (2015). Web Personalisation and Recommender Systems. *KDD*.
2. Huang, Y., Contractor, N. and Yao, Y. (2008). [online] Nosh.northwestern.edu. Available at: http://nosh.northwestern.edu/journals/2008Huang-5.pdf [Accessed 18 Apr. 2019].
3. Koehrsen, W. (2018). *Building a Recommendation System Using Neural Network Embeddings*. [online] Towards Data Science. Available at: https://towardsdatascience.com/building-a-recommendation-system-using-neural-network-embeddings-1ef92e5c80c9 [Accessed 24 Apr. 2019].
4. Rashid, A., Karypis, G. and Riedl, J. (n.d.). Learning Preferences of New Users in Recommender Systems: An Information Theoretic Approach. *KDD*.
5. Seo, J. (2018). *[Paper Summary] Matrix Factorization Techniques for Recommender Systems*. [online] Towards Data Science. Available at: https://towardsdatascience.com/paper-summary-matrix-factorization-techniques-for-recommender-systems-82d1a7ace74 [Accessed 24 Apr. 2019].
6. Zhang, F., Yuan, N., Lian, D., Xie, X. and Ma, W. (2016). Collaborative Knowledge Base Embedding for Recommender Systems. *KDD*.