ProjectReport

November 21, 2020

1 Abstract

!! TODO

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

!! TODO

2.3 New Idea

After learning about PageRank, and it's 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 a very 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 website 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`.

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 it's output is present in `ml-25m/clean_movies.csv`. Additionally, most of the movies in the final dataset were irrelvant, primarily because they were of other languages. Additionally, we could not process the entirety of the data. For example, a single 62000×6200 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) and 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 represent 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,

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

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 attempted 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 a very 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 due to the fact that there were very few total genres in the dataset. For director, language, and rating, the process of computing similarities were 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 different 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 later. Secondly,

!! TODO Harsh, explain about user-id.py

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.

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`.

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 process 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 and invert the matrices is also present there.

3.1.3 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

Library	Usage	Link
`numpy==1.19.4`	Various mathematical applications, including handling of very large matrices, matrix multiplication, and various other matrix operations	https://numpy.org/
`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/
`pandas==1.1.3`	For handling and processing of raw and processed data, in the form of `.csv` files	https://pandas.pydata.org/
`re`	Python Regular Expressions library for pattern matching while processing raw data	https://docs.python.org/3/library/re.html
`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/

Library	Usage	Link
`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 final result	https://graph-tool.skewed.de/
`bs4`	Web scraping	link
`lxml`	Web scraping	link

[ ]: