Creating a Movie Recommender Using the MovieLens Dataset

Edward White | ICS 491, Fall 2017 | University of Hawaii at Manoa

Introduction

Ever since I was a kid, I've enjoyed movies. I used to go to the movie theater nearly every weekend, and watch movies voraciously. While movies have been part of American culture basically as long as movies have existed and there was never a lack of big releases since my youth, technological advances in filmmaking technology have increased the quality and brought down the cost of creating content, leading to an explosion of high-quality film and television being made. Because of this, it can be difficult to keep up with all the best work, let alone all the gems that might appeal to you and no one else. In addition to advances in filmmaking technology, the advent of the internet made it possible to aggregate the opinions of millions of users about thousands of movies with increasing granularity, which has been led by companies such as Netflix, Hulu, IMDB, and many more. Because of the confluence of these two trends, I decided to make movie recommender.

Experimental Design

The experimental goal of this paper is fairly simple: Can big data techniques identify relationships between films, and provide reasonable recommendations?

In order to explore this question, we must:

- Identify and obtain a dataset with information that is connected to movie preferences
- Evaluate how the data is structured to decide what might reasonably be inferred, and how the data might need to be transformed
- Select which algorithms are likely to produce useful recommendations based on available data
- Decide how the algorithms should be implemented, and what technologies to use
- Evaluate the results of the algorithms

Selecting the Dataset

Movies are extremely high-dimensional, in the sense that there are many, many attributes that make them similar or different from each other, often within the same movie. Additionally, all these attributes are weighted differently, so a user could love one movie that has a set of attributes and hate another that has the very same attributes but might have emphasized one over the other. Because of the difficulty of measuring the similarity of movies directly, the better approach would be to let users tell us what movies are similar and which aren't, turning the user into a kind of principal component analysis that reduces all those high dimensions into a rating. As those ratings build up, we can use big data techniques to infer "people like this user feel this way about these movies."

Some obvious sources for large amounts of user ratings are Netflix, IMDb, and MovieLens. While IMDB has user ratings and more detailed information about movies, they have never published an official

dataset which meant obtaining the data could be time-consuming. Netflix has published an official dataset, which is fairly large at 100 million ratings, but is mainly limited to an anonymized user ID, a movie ID, and a rating. MovieLens on the other hand has several published datasets of various sizes, ranging as high as 20 million ratings, but they also do a good job of including some additional data about the movies that are being rated.

The 100k MovieLens dataset in particular got my attention, since it was the only dataset to include demographic data about its users, which meant that I would have more options to explore different relationships and possibly find new insights. The main concern was that, though the information was published in 2015, the data was based on ratings collected between September 19, 1997 and April 22, 1998, though a quick inspection of the data revealed that the ratings included many classics the average person would be familiar with, making evaluation a bit easier. Lastly, this set carries the additional advantage of being small enough to not require specialized hardware configurations to run the queries, though big enough to still require the use of big data tools to process.

About the MovieLens 100k Dataset

As the name implies, the MovieLens 100k dataset consists of 100,000 ratings by 943 users on 1,682 movies. Each user has rated at least 20 movies, and each user has simple demographic data associated to them. The dataset was cleaned by researchers at the University of Minnesota, removing any users that had not rated 20 movies or completed the demographic survey.

The dataset is broken out into various files, though the three important one are:

- u.data
    - 100,000 ratings by 943 users on 1,682 movies. Users and items are consecutively ordered from 1, and the data is randomly ordered
    - Tab separated values
    - Columns: user ID | item ID | rating | unix timestamp
- u.item
    - Information about the movies, with IDs matching the data file
    - Pipe separated values
    - Columns: movie ID | movie title | release date | video release date | IMDb URL | unknown | action | adventure | animation | Children's | Comedy | Crime | Documentary | Drama | Fantasy | Film Noir | Horror | Musical | Mystery | Romance | Sci-Fi | Thriller | War | Western
    - In the genre sections, 1 represents the movie being part of that genre, 0 indicates the movie is not part of that genre. Movies can be in several genres at once.
- u.user
    - Demographic information about the users, with IDs matching the data file
    - Pipe separated values
    - Columns: user id | age | gender | occupation | zip code

The other files in the dataset were information and metadata, such as a list of all the occupations, and the rest were various splits for testing and scripts for processing the data, none of which would be used for the project.

Evaluating the MovieLens Dataset

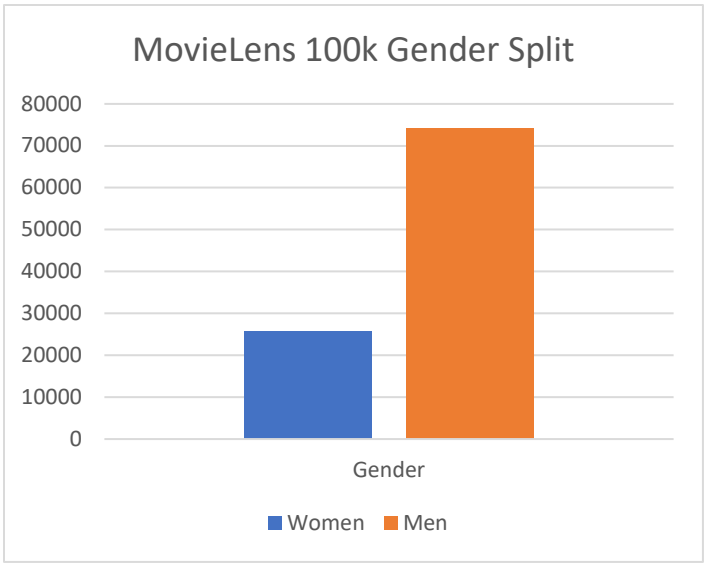
Because the MovieLens 100k includes demographic information, we're able to do some quick checks to see what biases might exist in the data. These checks are executed in Python, using the typical data toolkit of pandas for storing and manipulating the data in memory and manipulating, numpy for its math tools and parallelized operations, and MatPlotLib to export some of our charts.

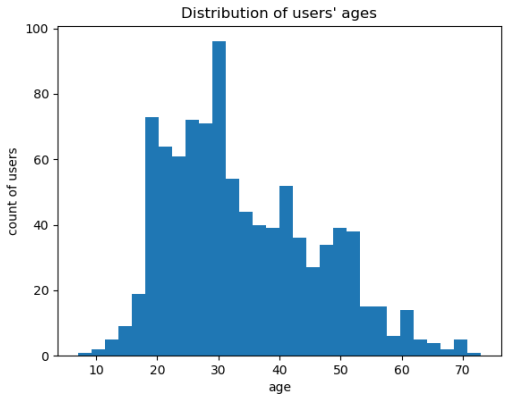
Despite the unusual file extensions, the MovieLens dataset is stored in plain text and uses separators that are easy to parse. Because of this structure, we are able to use the Pandas `read_csv()` function to read the dataset into memory. For each of the three files, the columns are defined as an array, and then passed into the read csv function, along with the filepath, separator, and encoding. For example:

```
12  r_cols = ['user_id', 'movie_id', 'rating', 'unix_timestamp']
13  ratings = pd.read_csv('ml-100k/u.data', sep='\t', names=r_cols,
14                        encoding='latin-1')
```

In order to make the data easy to work with, we then use the pandas `merge()` function to join all the data together in the variable "lens." Since the data has been pre-cleaned, we don't have to worry about blank spaces and other issues. This will allow us to use some more interesting pandas functions later.

For our first check, we count how many of each gender are in the dataset through grouping by gender and using the built in pandas `count()` function to figure out how much of each there is. Immediately we notice that men are very overrepresented in the data, with 74,260 males to 25,740, or a factor of about 3-to-1. This skew indicates the data may not generalize well when using certain algorithms, such as an undifferentiated regression analysis. Later analysis of this dataset will show there is a possible gender component to movie ratings, so it's wise to avoid data that relies on normal curves for accuracy.





The other standard check is age distribution. Pandas makes this extremely easy, since any numeric column is able to use the integrated `hist()` function which makes a histogram of the data, split into a specified number of bins. Pandas also has a MatPlotLib integration, so we just have to call `column.plot.hist(bins=n)` to visualize the age histogram, which reveals the age curve leans to a younger audience, but `skew()` and `kurtosis()` indicate the curve is (barely) within normal range.

While the data is technically normal, the moderate skewness of the data suggests we should take a look at whether there are significant differences with how each age group rates movies. To do this, we use the pandas `cut()` function to add an `age_group` column to our dataframe based on age buckets. Then we can use the `groupby('age_group')` function and call the `agg()` aggregate function on it, which passes in a dict with a column and an array of operations to do on that column, and returns a dataframe with columns containing the results. In this case, we want to use numpy's `size`, `mean`, and `std` functions to let us know exactly how big each group is, what their average rating is, and what the standard deviation for the rating is. The code and results are:

```
labels = ['0-9', '10-19', '20-29', '30-39', '40-49', '50-59', '60-69', '70-79']
lens['age_group'] = pd.cut(lens.age, range(0, 81, 10), right=False, labels=labels)
print(lens[['age_group']].drop_duplicates()[:10])

print(lens.groupby('age_group').agg({'rating': [np.size, np.mean, np.std]}))
```

Age Group	Rating Size	Mean	Standard Deviation
0-9	43	3.767442	0.996116
10-19	8181	3.486126	1.170631
20-29	39535	3.467333	1.154236
30-39	25696	3.554444	1.118818
40-49	15021	3.591772	1.091876
50-59	8704	3.635800	1.042335
60-69	2623	3.648875	1.006390
70-79	197	3.649746	1.075783

This data indicates the younger (and slightly overrepresented) population tends to rate movies slightly lower than their older peers, but the standard deviation suggests it's likely this difference is not statistically significant, and the difference is not significantly different in any case.

Out of curiosity, I checked what were the top 10 most rated and highest rated movies of all time in the dataset. To avoid outliers for highest, results are for movies with over 100 ratings.

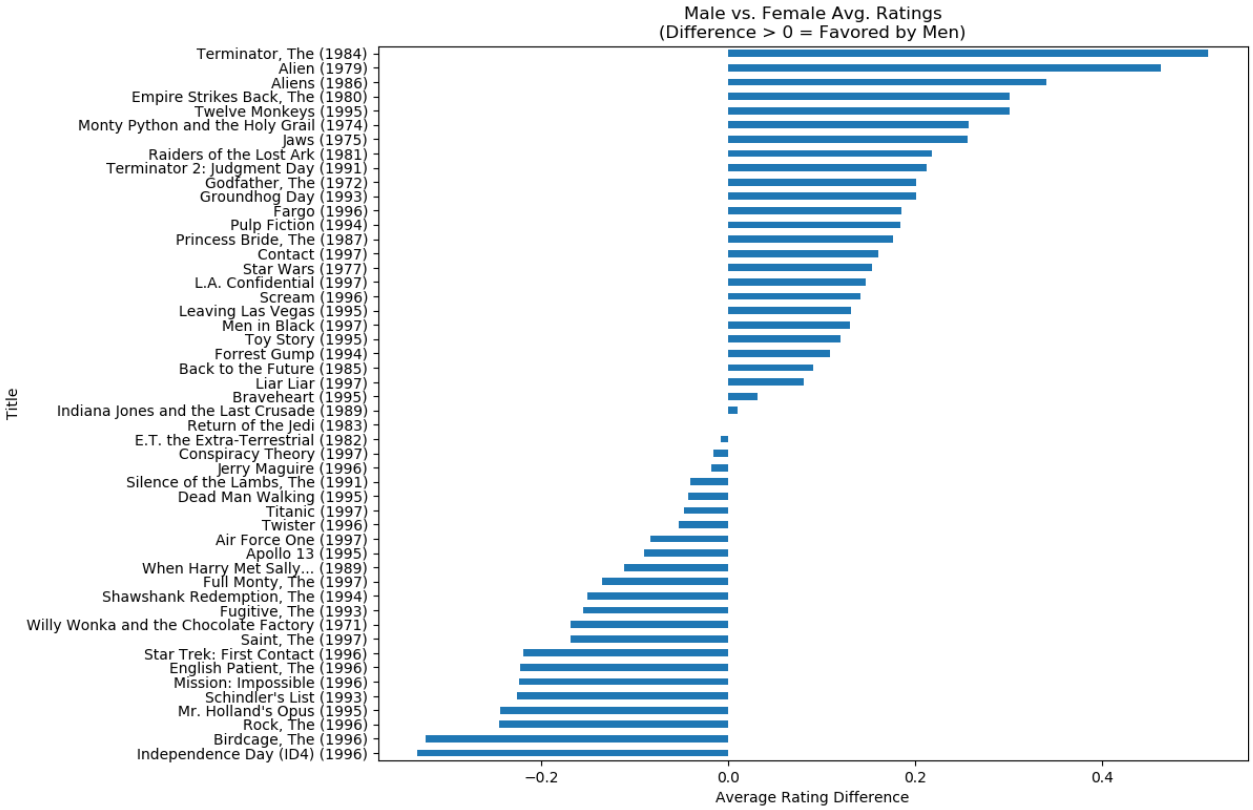
Title (Release Year)	Ratings
Star Wars (1977)	583
Contact (1997)	509
Fargo (1996)	508
Return of the Jedi (1983)	507
Liar Liar (1997)	485
English Patient, The (1996)	481
Scream (1996)	478
Toy Story (1995)	452
Air Force One (1997)	431
Independence Day (ID4) (1996)	429

Title (Release Year)	Ratings	Mean	Std
Close Shave, A (1995)	112	4.491071	0.771047
Schindler's List (1993)	298	4.466443	0.829109
Wrong Trousers, The (1993)	118	4.466102	0.823607
Casablanca (1942)	243	4.456790	0.728114
Shawshank Redemption (1994)	283	4.445230	0.767008
Rear Window (1954)	209	4.387560	0.712551
Usual Suspects, The (1995)	267	4.385768	0.825500
Star Wars (1977)	583	4.358491	0.881341
12 Angry Men (1957)	125	4.344000	0.719588
Citizen Kane (1941)	198	4.292929	0.846042

The top ten rankings seem to suggest that movies which were current at the time the data was collected have a higher number of ratings, but highest rated movies above the noise threshold are classic cinema.

Given the significant difference in the number of men and women in the dataset, it's prudent to check whether there is a gender component to how men and women rate movies. This check uses the pandas pivot table function to compare the differences in average rating between men and women in the top fifty most rated movies of the dataset.

```
69  pivoted = lens.pivot_table(index=['movie_id', 'title'],
70                             columns=['sex'],
71                             values='rating',
72                             fill_value=0)
73
74  print(pivoted.head())
75
76  pivoted['diff'] = pivoted.M - pivoted.F
77
78  pivoted.reset_index('movie_id', inplace=True)
79
80  disagreements = pivoted[pivoted.movie_id.isin(most_50.index)]['diff']
81  disagreements.sort_values().plot(kind='barh', figsize=[9, 15])
```



While the scale of the chart makes it seem like there is a large difference, it's much lower than the typical standard deviation encountered, so there is unlikely to be a statistical difference between how men and women rate movies. Nonetheless, there is a difference and at times it can be quite pronounced, so it's better to either get more women's ratings into the dataset or to use algorithms that are able to take into account the individual user as opposed to the system as a whole. This project takes the latter approach.

Based on this analysis, the data seems likely to give reasonable results for movie ratings pre-1998.

Selecting Algorithms and Processing Libraries

Based on our evaluation of the data and the problem, the main requirements are that the algorithms should be able to provide recommendations to individuals users, and should be able to cope with potential differences between men and women, which could be considered a subset problem. Two algorithms that seem likely to produce good results are FP Growth for Frequent Itemset Mining and Alternating Least Squares for collaborative filtering, both of which focus on a user's actions and compare them to users that are similar, and both of which provide discrete recommendations. This approach will additionally address the problem of the uneven distribution of men and women, since people who rate similarly will get recommendations from each other, so if women do rate differently on average, then they will receive ratings from other like-minded women.

Collaborative Filtering with ALS and Spark

Alternating least squares (ALS) is a relatively new algorithm, with the reference paper by Hu, Koren, and Volinsky having been published in 2008. A short heuristic for the concept is that the ALS algorithm can take two matrices that have a feature in common and use those to infer an approximation of a data point that does not exist in the matrix data frame. Because of this, it has become a popular algorithm for rating-based recommender systems, which often have lots of data that is structured in this way.

Another reason to choose ALS is because of its structural similarity to the Netflix Prize Dataset. As a bit of historical trivia, ALS is actually the algorithm which won the Netflix Prize in 2008 and 2009, and the winners included two of the three authors who published the paper describing the algorithm, as required by the contest. By 2009, the algorithm had achieved a 10.09% improvement over Netflix's Cinematch algorithm, with a regression mean square error of 0.8554 on the test dataset, which won them the grand prize.

For this algorithm, we only need the u.ratings file, which has the columns `userID`, `movieID`, `rating`, and `unix_timestamp`. For collaborative filtering, we need the first three.

The MovieLens rating dataset is well suited for use with ALS-based. Because each user has rated a movie, and each movie has been rated, the algorithm is able to use the `movieID`, `userID`, and `rating` fields to construct a "`userID X movieID`" matrix where there is a `rating` at each intersection, or a `0` where the user has not rated that movie in particular. This, in turn, allows us to construct the "`userID X f`" and the "`f X movieID`" vector lists that allow the ALS algorithm to estimate the projection of a high-dimensional vector that lies outside the data space and predicts the user rating.

While this dataset is relatively small, this algorithm requires $O(n^2)$ space to run its calculations, so libraries that are not designed for larger datasets tend to run into memory errors. Because of these issues, the Spark platform and its ALS library was selected to process the data. Because Pandas and Spark dataframes are interoperable, pandas was used to import the data into memory as described in the data analysis section, and then converted into a spark dataframe. The data was then randomly split 90/10, which were used as a training set and a test set to determine how well the algorithm worked. As described above, the userID and MovieID were passed in as user and item columns, and ratings where passed in as the intersection between the two.

The algorithm was evaluated using the root mean squared error method that was used to evaluate the ALS algorithm on the original Netflix Prize Dataset, resulting in an RMSE of 0.91 when using a regularization parameter of .1, which is not quite as close as the original ALS team was able to accomplish, but was well within the standard deviation of user reviews, so the results are likely to be better than random, and a manual inspection of the recommendations vs labels support that conclusion.

Frequent Itemset with FP Growth and Spark

Frequent Itemset mining is a valuable tool that reveals correlations between items across many transactions or a collection of sets, which allows analysts to understand what items tend to appear together, which may reveal non-obvious relationships between items. This is often used in marketplace basket analysis, but it can be generalized to work with any data that consists of sets that occur together in some sort of category. If the movieID and userID are thought of as transactions, the MovieLens ratings dataset could be used with the FP Growth algorithm to predict preferences in the following ways:

The algorithm requires the notion of "transactions" that contain items from which we can get itemsets. The Netflix Prize Dataset offers two categories which we can use to derive different insights:

- MovieID as transaction
    - The movieID is used as a transaction that contains all users that have rated it, possibly using a threshold (such as rating >= 4) to get only users that liked that movie.
    - Predicted insight: Users that frequently appear together like the same kinds of movies, so we can recommend movies where one user appears and the other doesn't.
    - Possible use: Should help with identifying which users are similar to each other, which could be used to power a recommendation engine that propagates movies among a network of users with high affinity for each other, which could help with proactive recommendations.
- userID as transaction
    - The userID is used as a transaction that contains all movieID entries rated by that user, possibly using a threshold (such as rating >= 4) to get only users that liked that movie.
    - Predicted insight: Movies that frequently appear together are likely related in some way, so when a user rates a given movie highly, a movie that has strong support and confidence can be suggested.
    - Possible use: Should help with identifying which movies are similar to each other, which could be used to power a recommendation engine that suggests movies similar to the one rated, which could help with reactive recommendations.

Given its similarity to the highly-successful Amazon model of "Customers who bought this item also bought" which powers its recommendation engine, we'll focus on the userID as the transaction analog. It also has the advantage of being computationally efficient, since the model only has to be checked each time a movie is rated, instead of propagating through a graph any time a critical mass of users discovers a new movie.

For this task, we'll also be using Spark since the FP Growth algorithm also has a high space complexity, and though the data set is small, it tends to grow faster than libraries such as pymining can handle it, while Spark is able to handle the data without requiring specialized hardware.

While the ratings dataset contains the information we need for a frequent itemset, it requires a fair bit of transformation to prepare it for use with the Spark implementation of the FP Growth algorithm, which requires an array of tuples, each of which has a transaction ID and an array of items. The need for extreme transformation makes reading the data directly into a pandas or spark dataframe inconvenient due to their immutable nature. Instead, the data was imported into an SQLite3 database so that the sets of needed data could be fetched and transformed in Python in the shape needed.

The first stage of transforming the data is to gather the users that will serve as our "transaction IDs." Because we want to recommend movies that users will like, we select all users that rated a movie 4 or higher on a scale of 5. Then we take that query and insert it into a for loop, gathering all movies that each user rated 4 or higher into an array. The userID and the array of movieIDs are then placed into a tuple, which is appended to an array. After this process is complete, the array is converted into a Spark dataframe with the columns labeled appropriately, and that is fed into the FP Growth algorithm with the movieID marked as the itemsCol.

The bigger the dataset is, the sparser it is going to be when it comes to frequency as a percentage of transactions, so the `minSupport` was set to 5% of data (which would still translate to 5,000 transactions in this dataset), but the `minConfidence` was set to 70% to cut down on potential results.

Manually reviewing the predictions indicated the results were reasonable. For example, someone who liked The Usual Suspects, Twelve Monkeys, The Empire Strikes Back, and Silence of the Lambs were predicted to be 96% likely to like Star Wars, and 92% likely to like Pulp Fiction. It just so happens that I love all those movies, and they are all considered classics. In another set, if you liked The Terminator, India Jones and the Last Crusade, The Empire Strikes Back, and Twelve Monkeys, you are 100% likely to enjoy Terminator 2, which is 100% correct for me.

The results very strongly imply that treating users as a transaction in a frequent item set is able to reveal complex relationships between movies that might seem unrelated, but have some sort of common underlying qualities that attract a certain kind of user, and which might be difficult to quantify with a purely mathematical and modeling approach.

Conclusions

Despite some of the possible shortcomings of the dataset, Alternating Least Squares and Frequent Itemset algorithms both were able to work around these issues and showed a lot of promise as the underlying software for movie recommender systems, with high accuracy in both mathematical and qualitative testing, which is not surprising given that at least the ALS algorithm was designed with this specific purpose in mind. The frequent itemset approach, while not specifically designed for this problem, could have been expected to perform this well given its use in other consumer recommender systems, especially those used by Amazon. While the algorithms were previously held back from use in large enterprise applications by their high compute requirements, the ability to do these operations in parallel, either across several processors or several clusters, brings the use of these algorithms into the realm of being realistic to use in large-scale systems.