CSE 701

A Graph based recommendation system using content-assisted collaborative filtering

Ankit Kapur: 5013-3149
Harishankar Vishwanathan: 5013-4706

Introduction

Although recommendation algorithms using content based and collaborative filtering techniques exist, viewing data as a graph allows one to slice and dice the data in many ways in order to bring out complex implicit relationships. To that end we have tried to create a movie recommendation system that models user information, movie data and genres as nodes in the graph, and represents the relationships between these nodes (such as ratings given to movies, and genres that a movie has) as edges.

We first began with a basic collaborative filtering technique for generating recommendations for a user, by traversing each of the movies watched by the user, and then traversing further to discover other users who had also watched these movies. We then get the movies watched by these other users, filter out duplicates, and sort by the number of paths leading to the movie. The top n movies in this sorted movie list are then presented to the user as recommendations.

We then improve upon this simple recommendation model, by incorporating content into the recommendations in addition to collaborative filtering. We do this by factoring in genres to our recommendations, recommending only those genres which are already watched by a user.

For modelling the data as a graph, we have used the graph database Neo4j, and to operate on the graph, we use Gremlin as a medium, to make it easier to traverse the graph. With the backing database, and using Gremlin's easy traversing capabilities, it becomes easy to provide recommendations using collaborative filtering techniques, and content-based hybrid techniques.

Our design for this project is simple, involving only 3 types of nodes and 2 types of edges for relationships, but the graph model is a scalable base for a full-scale recommendation system. For a complex system with numerous types of complex relationships and entities, the power of the graph traversal pattern can be harnessed to generate more meaningful recommendations.

Tools and Dataset

For our purpose, we have used the MovieLens dataset. GroupLens Research has collected and made available rating data sets from the MovieLens web site (http://grouplens.org/datasets/movielens/). The dataset was released in February 2003, though it collected over various periods of time. It contains a total of 1 million ratings and 465,000 tag applications applied to 4000 movies by 6000 users,

movies.dat
Each row of this file contains the movie's ID, name, and genre

```
519::Ace Ventura: When Nature Calls (1995)::Comedy
520::Money Train (1995)::Action
521::Get Shorty (1995)::Action|Comedy|Drama
522::Copycat (1995)::Crime|Drama|Thriller
523::Assassins (1995)::Thriller
```

If a movie belongs to multiple genre, the genre field is delimited by pipe symbols.

users.dat
Structure of users' data has the following structure:

```
22::M::18::15::53706
23::M::35::0::90049
24::F::25::7::10023
25::M::18::4::01609
26::M::25::7::23112
```

Each row of the raw file has 5 columns: userId, gender, age, occupation, and zipcode. We have ignored the zipcode field.

ratings.dat
Contains the ratings that a user provided for the movies they watched. Each row of the raw file has 4 columns: userId, movieId, stars (1-5 rating scale), and timestamp. We have ignored the timestamp field.

```
1::1097::4::978301953
1::1721::4::978300055
1::1545::4::978824139
1::745::3::978824268
1::2294::4::97882429
```

Tools

We have used **Neo4j** for the graph database. Neo4j can serve our purpose to efficiently store, handle and query highly connected elements in your data model. With a powerful and flexible data model you can represent your real-world, variably structured information without a loss of fidelity. The property graph model is easy to understand and handle, especially for object oriented and relational developers.

For traversing the graph, we use the Gremlin graph traversal language which works over graph databases, like Neo4J. Gremlin is a style of graph traversal that can be used in various JVM languages, we use Java and Groovy.

Building the graph

Schema

The schema for the graph database we created is shown in the figure below.

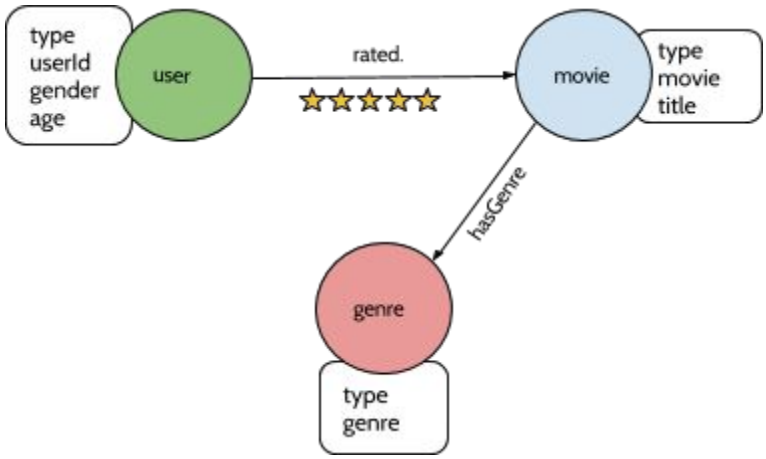

Fig. The Scheme for the graph database

The ***rated*** edge is unidirectional, since a user *rates* a movie, and represents an **m:n relationship** between the two vertices since a user can rate multiple movies, and a movie can be rated by multiple users.

The ***genre*** edge also represents a unidirectional m:n relationship since a movie can have multiple genre and a genre can belong to multiple movies.

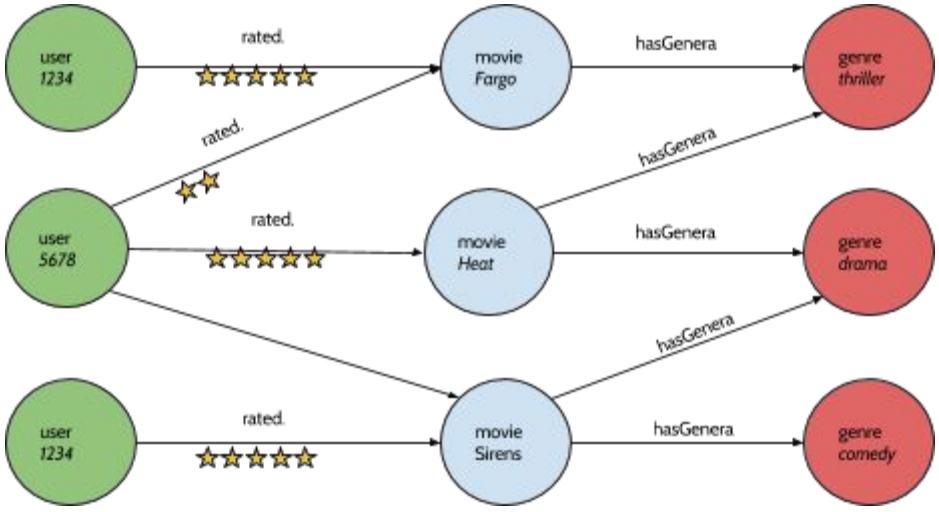

Fig. An example of the graph instance.
The User vertices are represented in green, Genre in red, Movies in blue.

Vertices:
We extract data from the MovieLens dataset to create 3 types of vertices: User, Movie, Genera, all of which are quite self explanatory.

Edges:
Genre that a movie belongs to are represented as directed edges between Movie and Genera vertices. Users' occupations are represented as directed edges between User and Occupation vertices.

Parsing
We add the nodes related to the different *movies* by parsing movies.dat. This is also where we create the *genre* nodes. We add the nodes related to the *users* vertices by parsing users.dat. Using ratings.dat, we add edges between the nodes, creating relationships between them. The *rated* edge goes from A user to a movie (if the user has rated the movie) and the property at the edge gives the actual rating given (0 through 5 stars). The *hasGenre* edge goes from a *movie* node to the *genre* nodes it "belongs" to.

Generating recommendations

Now that we have populated the graph database, we can query it and use it for recommendations. First we try collaborative filtering, where we try to correlate the ratings behaviour of users in order to recommend the favourites of one user to another similar user. We then extend our idea to include content as well.

Using Collaborative Filtering

To give recommendations for a user we do the following:

1. Start with the user vertex, say with *userID:* 1234, for whom recommendations are to be generated. We will refer to this user as the seed user.

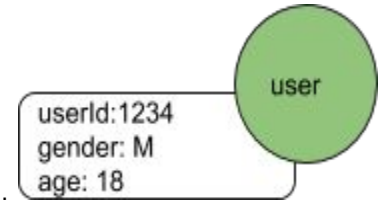

Fig. The 'seed' user vertex

2. Get all the movies the user has given of rating greater than 3, as a list.
   This is equivalent to:
   a. Get all the *outgoing* edges from the user.
   b. Filter so that all the edges have the *stars* property > 3
   c. Get the movie vertices corresponding to the edges, and add it to the *userRatedMoviesList.*

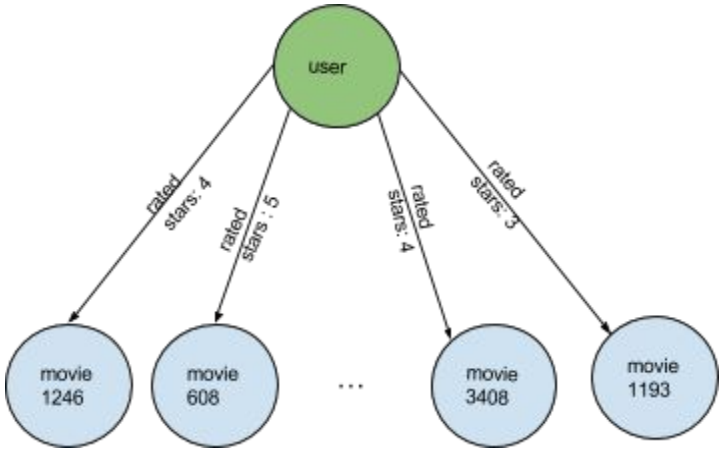

Fig. The *userRatedMovies* list contains all the movies rated by the user (blue vertices)

3. For each movie in the *userRatedMoviesList*, find other movies which were given 3 or more stars, by users who gave 3 or more stars to the given movie: basically find movies "co-rated" to the given movie.
   We will refer to the final list of recommendations as *recommendationList*.

   This is equivalent to:
   For each movie in the *userRatedMoviesList*:
   i. Get the incoming "rated" edges
      Each of these edges correspond to ratings given to a movie watched by the seed user.
   ii. Filter out those edges whose star property is less than 3
   iii. Get the movie vertices corresponding to the edges
   iv. Put the movies into *recommendationList*.

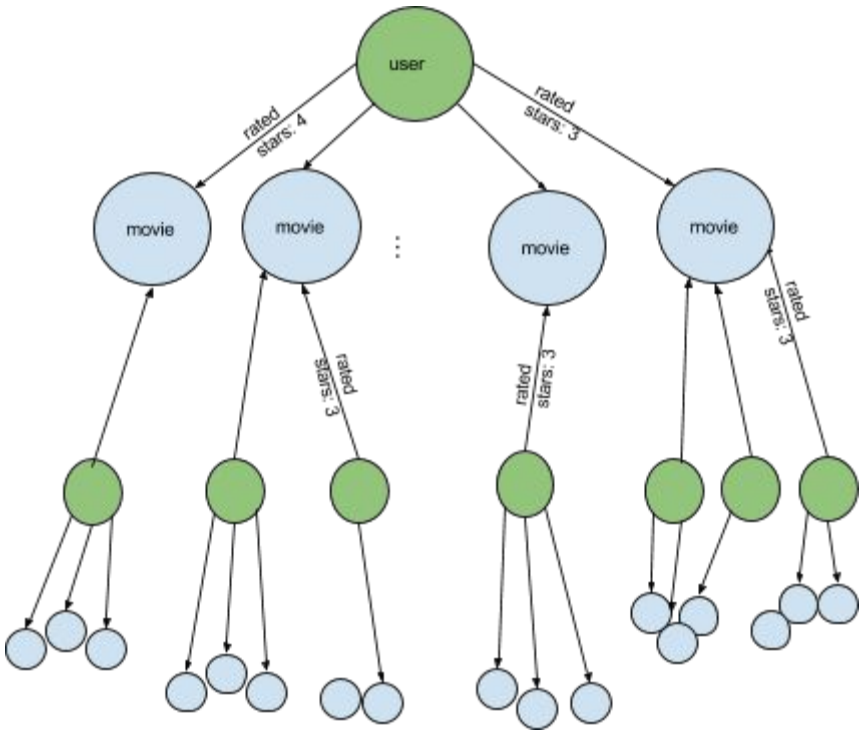

Fig. Recommendation process.
The blue movie vertices on the bottom appear in the *recommendationList*

4. Now that we have the recommendations, we need to have a way to rank them. Here we note that, in step 3, given a movie in *userRatedMoviesList,* there will be many high rated paths from it to other movies, many (most) of these paths will lead to the same movies. It is possible to use these duplicates for the ranking mechanism. The higher number of paths to a movie starting from the given movie in *userRatedMoviesList*, the higher the movie is co-rated to the given movie. To do this we:

   a. Find the number of paths from each movie in *userRatedMoviesList* to co-rated movies, and store the count in a map: <movie, count>
   b. Return the map, sorted by count as the final recommendation.

Adding content based recommendations

5. We can use genre information to further curate our recommendations. To do this, when we find movies co-rated to the movies in *userRatedMoviesList* , we filter out those movies which do not have the same genre(s) as the given movie. This is done in the following way:

    a. Start with step 4.
    b. Find the number of paths from each movie in *userRatedMoviesList* to co-rated movies, and store the count in a map <movie, count>, but this time, choose only those movies which share a genre with the movie from *userRatedMoviesList*.
    c. Return the map, sorted by count as the final recommendation.

Results

1. We start with userID 1234:

```
gender=M
type=User
userId=1234
age=18
```

2. The *userRatedMoviesList,* is the list of movies given more than 3 stars by the user 1.

```
{movieId=1252, type=Movie, title=Chinatown (1974)}
{movieId=589, type=Movie, title=Terminator 2: Judgment Day (1991)}
{movieId=3948, type=Movie, title=Meet the Parents (2000)}
{movieId=1270, type=Movie, title=Back to the Future (1985)}
{movieId=913, type=Movie, title=Maltese Falcon, The (1941)}
{movieId=922, type=Movie, title=Sunset Blvd. (a.k.a. Sunset Boulevard) (1950)}
{movieId=930, type=Movie, title=Notorious (1946)}
{movieId=2291, type=Movie, title=Edward Scissorhands (1990)}
{movieId=1673, type=Movie, title=Boogie Nights (1997)}
{movieId=3435, type=Movie, title=Double Indemnity (1944)}
{movieId=260, type=Movie, title=Star Wars: Episode IV - A New Hope (1977)}
{movieId=3481, type=Movie, title=High Fidelity (2000)}
{movieId=2858, type=Movie, title=American Beauty (1999)}
{movieId=1127, type=Movie, title=Abyss, The (1989)}
{movieId=3865, type=Movie, title=Original Kings of Comedy, The (2000)}
{movieId=1196, type=Movie, title=Star Wars: Episode V (1980)}
{movieId=1198, type=Movie, title=Raiders of the Lost Ark (1981)}
{movieId=3897, type=Movie, title=Almost Famous (2000)}
{movieId=2186, type=Movie, title=Strangers on a Train (1951)}
{movieId=1997, type=Movie, title=Exorcist, The (1973)}
{movieId=1210, type=Movie, title=Star Wars: Episode VI (1983)}
{movieId=2028, type=Movie, title=Saving Private Ryan (1998)}
```

3. (and 4)
   Now for each movie in the above *userRatedMoviesList*, we find co-rated movies. The higher number of paths to a movie starting from the given movie in *userRatedMoviesList*, the higher the movie is co-rated to the given movie. The following is the aggregated list of the final recommendations, obtained by sorting the movies according to the count of the number of paths leading to it.

```
Star Wars: Episode IV - A New Hope (1977)=16016
Star Wars: Episode V - The Empire Strikes Back (1980)=15800
Matrix, The (1999)=14751
Raiders of the Lost Ark (1981)=14271
American Beauty (1999)=14251
Silence of the Lambs, The (1991)=14241
Fargo (1996)=13510
Godfather, The (1972)=13497
```

```
Sixth Sense, The (1999)=13269
Shawshank Redemption, The (1994)=13222
```

5. When we consider the content (i.e. the genre), as per step 5, we get the following results. This as mentioned before, is obtained by considering as a movie co-rated to another movie, only if they share the same genres.

```
Star Wars: Episode V - The Empire Strikes Back (1980)
Princess Bride, The (1987)
Matrix, The (1999)
Star Wars: Episode IV - A New Hope (1977)
Braveheart (1995)
Godfather, The (1972)
Terminator, The (1984)
Saving Private Ryan (1998)
Star Wars: Episode VI - Return of the Jedi (1983)
Alien (1979)
```

These are the results we can finally present to the *seed* user as **top 10 recommendations**.

Conclusion

The recommendation system we have created is capable of generating recommendations based not only on collaborative filtering but also on content. Though our model is simple in nature, in that it only uses genre information as *content*, but is capable of generating much more sophisticated and meaningful recommendations if other *content* information is used as well. Movie metadata, such as the movie's directors (with edges for *directedBy*), actors, award categories, nomination categories, can be used as content information by creating filters in the way we have done. Additionally, User metadata, such as their age, gender, occupation etc. can also be used. For example, age can be used to filter out PG-13, and R movies and adult content out for under-age users. Another use-case for this system can be answering the question, *what movies do those users prefer who **don't** like the same movies as me.* This can be done by using a < 3 rating filter on edges instead of > 3 the way we have done.

References

1. A Graph-based Recommender System for Digital Library:
   http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.66.5744&rep=rep1&type=pdf
2. A Recommender System to Support the Scholarly Communication Process:
   http://arxiv.org/pdf/0905.1594v1.pdf
3. Graph Based Recommendation Systems at eBay http://www.slideshare.net/planetcassandra/e-bay-nyc
4. Studying Recommendation Algorithms by Graph Analysis
   http://people.cs.vt.edu/~ramakris/papers/receval.pdf
5. Real-Time Recommendation Engine with Neo4j
   http://neo4j.com/use-cases/real-time-recommendation-engine/
6. MovieLens dataset: 1 million ratings from 6000 users on 4000 movies
   http://grouplens.org/datasets/movielens/
7. Gremlin and graph traversal tutorial:
   http://markorodriguez.com/2011/09/22/a-graph-based-movie-recommender-engine/
8. Groovy:
   https://learnxinyminutes.com/docs/groovy/
   http://www.ibm.com/developerworks/library/j-pg04149/
   http://docs.groovy-lang.org/
9. Gremlin:
   https://github.com/tinkerpop/gremlin/wiki/Using-Gremlin-through-Groovy
   http://s3.thinkaurelius.com/docs/titan/0.5.4/gremlin.html
10. Neo4j:
    http://neo4j.com/developer