Comparison of SVD and Graphical approaches for Recommendation System

Saurabh Agrawal (agrasaur@iu.edu)

1) Objectives and significance

This project compares different approaches for performing collaborative filtering. I compared several recommendation systems algorithms by accuracy, practical run-times, etc.

We are living in the age of information overload. There is so much information that it gets hard to find the information one is trying to find. There have been some experiments where customers refused to buy when given too many choices. This phenomenon is called 'The paradox of choice'.

Recommendation systems try to alleviate this problem by finding items that are supposed to be relevant to the customers. This not only helps increase the sales, but also helps customers to find relevant products quickly. Netflix suggests that 75% of the content watched comes from its recommendation engine. Similarly, some experts believe that 35% of Amazon's sales are generated by the recommendations.

A great amount of work is going on to build better recommendation systems. People from both the academia and industry are striving to build systems that make better recommendations. Netflix prize was an open challenge for building a recommendation system that decreases the error by 10%. This challenge accelerated the research on recommendation systems.

2) Background

The emergence of online marketplaces has had a huge impact on the consumers, providing them access to a huge variety of products. Although, the online commerce has emerged as a billion-dollar industry, it has made it more difficult for customers to find the products they need. One of the well employed solutions to this problem is recommendation systems, which provide automated suggestions to customers.

The recommendation problem can be defined as predicting the rating of a user for items, based on their responses to other items, responses from similar users to the same item, etc.

2.1) Formal definition of the problem

Suppose,

U: the set of all users in the system

I: the set of all items in the system

R: The set of ratings recorded in the system

S: Possible values of all ratings. Eg: {1, 2, 3, 4, 5}, {like, dislike}, etc

r_{ui}: Rating made by user u for item i

U_i: Users who have rated item i

I_u: Items rated by user u

I_{uv}: $I_u \cap I_v$

The task is to learn a function f: U X I $\rightarrow$ S that predicts f (u, i): Rating made by user u towards item i.

2.2) Commonly used approaches

2.2.1) Content based recommendation systems

Content based recommendation systems try to identify common features of items that received a good response from a user u, and then recommend new items to u that share these features. For example, a user who liked a horror comedy could be recommended other horror comedies. Apart from identifying the features of items, we also need to maintain a preference profile for each user.

These systems try to recommend the most similar items to users to the ones they liked. These systems suffer from three main problems:

- Identifying features in items might require manual expertise (thus increasing costs). For example, a company might have to employ human experts to classify movies or songs.
- It might be difficult to make recommendations when there is not much information about the user.
- These systems might end up making over-specialized recommendations. Thus, these systems might never make recommendations that are dissimilar but might be interesting to the user.

2.2.2) Collaborative filtering-based recommendation systems

The key idea in collaborative filtering approaches is that rating of a user u for a new item i is mostly going to be like that of another user v if u and v have rated other items in a similar way. Similarly, u is likely to rate two items i and j in a similar manner if other users have given similar ratings to these two items.

The biggest benefit with Collaborative filtering approaches is that these are domain agnostic, thus reducing costs. Some of the other benefits are recommending diverse items, simplicity, etc.

2.3) Previous work

There has been a lot of work on recommendation systems. The interest in recommendation systems suddenly piqued after the announcement of the Netflix prize. The current approaches vary from manually assigning tags to items, performing matrix factorization, to using Restricted Boltzmann Machines to predict the best ratings. However, the best systems usually work using a hybrid approach by forming an ensemble of various predictors.

2.4) Why is this problem interesting?

Today, recommendation systems are not only used in e-commerce industries, but also in healthcare and finance industry. Also, this problem has been approached differently by people from different domains

like Computer Science, Statistics, Electrical engineering, etc. Recommendation Systems not only need to be accurate, but they need to be fast enough to give results in real-time.

Aside from this, I have always found recommendation systems and graph theory to be fascinating. This project lies at the intersection of both.

3) Methods

I only worked on collaborative filtering-based recommendation systems in this project, therefore some important collaborative filtering methods are discussed as follows

3.1) Components of neighborhood methods

3.1.1) Rating normalization: Every user has their own personal scale to rate an item. Some users habitually give higher ratings to all movies, while others do not. We could normalize the users' ratings to negate this effect. Following are the methods to perform rating normalization.

- **Mean centering**: In this method we find mean of a user's ratings and subtract it from all the ratings. For example: If a user u gave the ratings 3 and 5. Then their mean centered ratings will be -1 and +1.
  $MC(r_{ui}) = r_{ui} - \text{mean}(r_u)$
- **Z-score normalization**: Apart from the mean centering, z-score normalization also considers the spread in a user's rating scales.
  $ZN(r_{ui}) = (r_{ui} - \text{mean}(r_u)) / sd(r_u)$ where sd() is the function to find standard. deviation

3.1.2) Similarity weight computation: Once we have normalized the ratings, we need to find similar users to predict ratings of unrated items for a user. We can use different methods like Cosine similarity, Pearson coefficient, Spearman Rank Correlation, etc.

3.1.3) Neighborhood selection: After we have computed similarity between all pairs of users, we need to find the closest neighbors. Following are the ways that could be used to find such neighbors.

- **Top N filtering:** In this approach, we select a pre-defined number of closest users.
- **Threshold filtering:** In this approach, we fix a threshold for the similarity value, and select all the users that have a similarity value less than the threshold.

3.1.4) Validation: After building the recommendation system, we perform cross-validation to compute the validation error. There are different ways to compute the validation error.

- **Mean Absolute Error:** In this approach, we sum the absolute difference between the predicted and test ratings. Then, we divide this sum by the total number of ratings.
- **Root Mean Squared Error:** In this approach, we find the sum of squared distances between the predicted and the test ratings. Then, we divide this sum by the total number of ratings.

Basic algorithm for neighborhood methods

(i) Perform normalization on ratings matrix if required
(ii) Find similarity between all pairs of users and store it in a similarity matrix

(iii) Now, if we need to predict rating r_{ui} (rating for item i by user u), we could find the closest users who have rated the item i.
(iv) Next, we could average these ratings (or weighted average) to find the predicted rating
(v) Find the validation error

3.2) Single Value Decomposition (SVD)

SVD is a method in linear algebra to factorize a matrix into 2 component matrices. This decomposition is done using an iterative algorithm that initializes the component matrices with random values and tries to reduce the error at each iteration.

Basic algorithm for SVD

(i) Perform normalization of ratings if required
(ii) Decompose the [U X I] matrix into 2 matrices A and B of dimensions [U X n] and [n X I] respectively, where n is the number of latent features.
(iii) Multiply A and B to get the prediction matrix
(iv) Find the validation error

3.3) A graph-based approach

The biggest problem with the approaches mentioned above is these do not perform well with sparse data. For example, top-n filtering requires that there should be n similar users. We might not have n similarities if there are many items.
This problem could be alleviated by considering longer relationships. For example, we need to predict r_{ui}: rating of user u for item i. Further, let's suppose that no user u is similar to user v1 who is similar to user v2, and user v2 liked item i. We can still infer that user u might like item i.
I worked on a graph-based approach mentioned in [1].

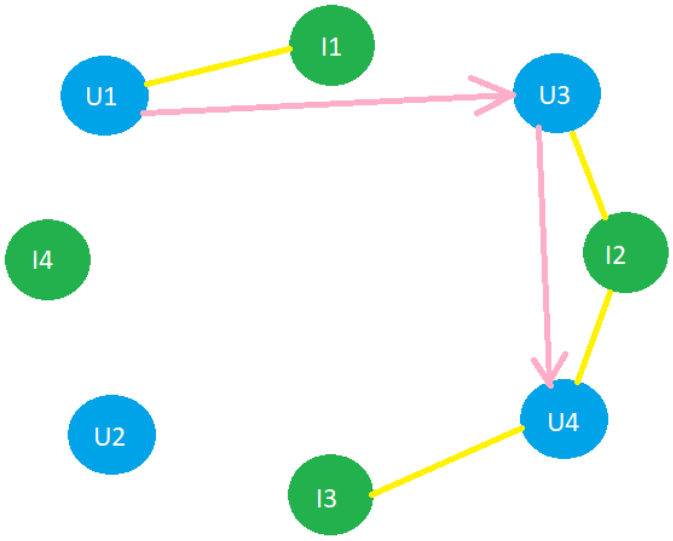
3.3.1) Terminology

Horting: Horting is a relationship between two users that is satisfied if these users have rated similar items.
We say that user u horts user v if either $|I_{uv}| >= alpha$, or $|I_{uv}|/|I_u| >= beta$
Horting is an asymmetric relationship.

Predictability: Predictability is a stronger relationship than horting that requires the ratings of the two users to be similar.
We say that user v predicts user u if u horts v and

$$\frac{1}{|Iuv|}\Sigma\ |r_{ui} - l\ (r_{vi})| <= gamma$$

Where *l* is a linear transformation function that maps the ratings between the two users.
Finally, the rating is evaluated by transforming the rating of the user who rated the item



Suppose, we need to predict rating (U1, I3), we can see that I3 has been rated by U4 (yellow line). Further, we can see that there is a predictability path (pink line) between U4 and U1 (U3 predicts U1 and U4 predicts U3). We can find the predicted rating (U1, I3) by finding the linear transformation of rating (U4, I3). If there are multiple such paths, we can average these ratings.

3.3.2) Algorithms

Algorithm for building the graph

(i) Read the data from the file in memory

(ii) Construct a graph using 2 different types of nodes: user nodes and item nodes. Create all the nodes in the graph

(iii) Next, create a rating relationship (edge) from a user u to the item i if u rated i. Also store the rating as an attribute to the relationship

(iv) Now, for each pair of users u1 and u2, check if u2 predicts u1. If it does, create a predicts relationship (edge) u1 -> u2. Store s, t, and the distance attributes in the relationship

Algorithm for finding predicted rating

(i) Suppose, need to predict rating (u, i)

(ii) Find the list of all the users who rated item i. Let this list be called raters.

(iii) Now find the shortest paths between u and each user in raters.

(iv) Compute the rating for user u via each shortest path

(v) Find the average rating of all these ratings. This is the predicted rating.

(vi) Compute the validation error

4) Results

I used MovieLens 100k as the dataset. I performed 5-way cross-validation to compute the mean validation error. I used my HP laptop (i5 3rd generation 2.4 GHz, 8 GB RAM) to run these programs.

4.1) Results for Neighborhood methods

(i) Average weight of top 25 similar users

Validation errors

Root Mean Squared Error: 1.22

Mean Absolute Error: 1.035

Runtimes

Total time: 15 minutes for 5-way cross-validation

(ii) Weighted average of top 25 similar users

Validation errors

Root Mean Squared Error: 1.243

Mean Absolute Error: 1.032

Runtimes

Total time: 15 minutes for 5-way cross-validation

4.2) Results for SVD

Validation errors

Root Mean Squared Error: 2.301

Mean Absolute Error: 2.006

Runtime

Total time: 90 minutes for 5-way cross-validation

4.3) Results for the Graph algorithm

Max path length: 2

Validation errors

Root Mean Squared Error: 1.131

Mean Absolute Error: 0.865

Runtimes

Graph building: 45 minutes

Predicting: 6 hours for performing 5-way cross-validation

5) Conclusions

We can see that the graph algorithm gives the lowest validation error amongst all the other algorithms. However, it is also the slowest. I tried running the graph algorithm for Max path length = 3, and it could only make 15,000 predictions in 8 hours. But, the graph algorithm is also the one that could be parallelized the easiest. We can simply process each path in an embarrassingly parallel manner.

SVD is one of the most used algorithm for a simple recommendation system, however, it did not give very accurate predictions with my implementation.

The neighborhood methods although being simple works surprisingly well.

It will be really cool to test the ensemble of these methods and test the validation errors.

6) References

[1]: Horting Hatches an Egg: A New Graph-Theoretic Approach to Collaborative Filtering (Charu C. Aggarwal, Joel L. Wolf, Kun-Lung Wu, Philip S. Yu)

[2]: Recommender System Handbook (Francesco Ricci, Lior Rokach, Bracha Shapira, Paul B. Kantor)

[3]: https://www.lynda.com/Python-tutorials/Introduction-Python-Recommendation-Systems-Machine-Learning/563080-2.html