The Happy Reading Club

Programming and Algorithms II, Degree in Bioinformatics
November 2017

1 Problem statement

A group of friends decides to write a program to share their opinions and recommendations on the books they read.

To simplify this exercise, books and readers are identified by their names only (strings). In a real case, we would need classes with other information (email, publisher of the book...).

After thinking, the operations desired for a class ReadingClub are (add the `self`'s):

- `__init__`: create an empty Reading Club
- `rate(reader,book,rating)`: reader is a reader name (a string); if not seen before, s/he is added to the club. book is a book name (a string); if not seen before, it is added to the club; rating is a value between 0 and 5 (may have decimals), indicating how much the reader liked the book; 0 is not at all, 5 is passionately. If the reader already read the book, the old rating is replaced with the new one.
- `who_read(book)`: returns a list of pairs with names of readers that have read this book and their ratings. Empty if no one has read the book.
- `what_read(reader)`: returns a list of pairs with the books read by reader and his/her ratings. Empty if there are no ratings by this reader.
- `recommend(reader)`: returns a book that reader reader may like, on the basis of the ratings. If no reasonable recommendation can be found, it returns the empty string.

Three observations:

- Conceptually, all that we are doing with the first operations is maintaining a *rating matrix* M, where rows are readers, columns are books, and $M[r, b]$ is the rating of book b by reader r, with some extra value such as -1 to indicate "r has not rated b" However, this matrix is very sparse because on average every reader has only read a very small fraction of the books, so most of the matrix entries would be -1, and this wastes memory. We want a representation that uses memory proportional to the number of ratings entered so far.

- There are many ways to do recommendations. One is to always give the reader the book with the best average rating among those that s/he has not read. Try doing that version first. But this would give readers whose interest is philosophy or bioinformatics the (highly popular) post-retirement memoirs of Lady Gaga, which they may not like. A more personalizing method known as "Cooperative Filtering" is described in the Appendix.

- Note that we only have the ratings to do the recommendation. In real life, we have more info, such as the book author or the genre. You may want to recommend the next book by an author s/he previously liked, or books on the same topic that some s/he liked before.

How would you change the class (or do a subclass) if now you have to provide this operation:

- `average_rating(book)`: returns the average rating of a book; -1 if not yet rated.

Bonus Track: Collaborative Filtering (CF)

The idea is simple: recommend me the books that 1) I did not read yet and 2) were rated highly by people with my same tastes. You know that someone has your same tastes if *on the set of books that we both have rated*, our ratings were similar.

This is what we do all the time: we look for recommendations from friends, blogs, youtubers, ..., whose advice we have liked in the past. Now

we have to turn this into mathematics so that we can program it – a program is nothing else than a mathematical formula.

Let us denote by R_x the set of books rated by any reader x and, for two readers x and y, $R_{x,y} = R_x \cap R_y$, the set of books that both x and y have read.

We can look at the vectors of ratings of x and y and compute how similar they are by simply taking their Euclidean distance. The similarity of x and y (how close their tastes seem to be based on their ratings) is:

$$\frac{1}{|R_{x,y}|} \sum_{b \in R_{x,y}} (R(x,b) - R(y,b))^2,$$

where $R(x,b)$ is the rating of book b by reader x. You can also use absolute value instead of squaring. Note that this is undefined $0/0$ if there is no books shared by x and y.

Then, in order to recommend a book to x we want to:

- compute $sim(x,y)$ of x to every y;
- find the book b in $R_y - R_{x,y}$ high the highest rating by y;
- recommend b to x.

Yes, this is costly (you need to visit all ratings, which is slow, or have all the similarities among all pairs of users recomputed, and update them when there is a new rating, which makes rating slow and uses memory quadratic in the number of users).

Also, many things can go wrong:

1. what if x has not yet read any books? Or not any books that anyone else has read? (This in particular happens at the start of the process when no one has read any books. This problem is called the "coldstart problem" of collaborative filtering and is, with its computational cost, its major problem. You want to resort to other strategies, like using the additional information (author, genre...) or recommending the book best rated in average, etc.)

2. what if you compute the most similar user y, and all books read by y already have been read by x? Or if all the books read by y but not by x were rated very low by y? Perhaps you want to try another y that, even if less similar to x.

Bonus bonus track: This basic implementation of CF can be improved in several ways.

One, is taking into account that some similarities are more reliable than others. If we compute $sim(x, y)$ on the basis of 10 common books, we are more confident that if x and y have read only 1 book in common.

Second, for someone whose average is 1.5, 3 is a good rating; for someone whose average is 4.5, a 3 is a bad rating. To compute similarities, you probably do not want to consider absolute ratings, but the ratings adjusted by the average of the reader.

Finally, a practical problem is to protect our book from spamming or malicious ratings: authors or publishers creating many fake readers and ratings to upvote or downvote specific books, or just to sabotage the system.