Project 1: Book Recommendations

CS 1410

Background

When buying things online you have probably noticed that you are often presented with other items that “you might also like” or that “other customers also bought”. In this project you will recommend books to a reader based on what other readers with *similar tastes* have liked.

Netflix awarded one million dollars to the winners of the Netflix Prize. They asked competitors to find an algorithm that would perform 10% better than their own algorithm. Making good predictions about people's preferences was that important to this company. It is also a very current area of research in **machine learning**, which is part of the area of computer science called **artificial intelligence**.

Objective

In this assignment you will discover **book recommendations** for readers based on other readers with *similar tastes* in books. (See the sample execution below on page 3.) The purpose of this assignment is to use common Python **data structures** (lists, dictionaries, sets) and **file input and text processing operations** in a program that is a little *larger* than any you may have encountered up to this point. It is important that you understand the different parts of the program and plan ahead of time how you will implement them. A suggested order for design and development appears later on in this document.

Data

There are two input data files used in this project.

First, there’s a list of books in “author,title” format in the file *booklist.txt* in Files/Projects/Project 1 on Canvas:

```
Douglas Adams,The Hitchhiker's Guide To The Galaxy
Richard Adams,Watership Down
Mitch Albom,The Five People You Meet in Heaven
…
```

There is also a file there with user ratings for each book (*ratings.txt*):

```
Ben
5 0 0 0 0 0 0 1 0 1 -3 5 0 0 0 5 5 0 0 0 0 5 0 0 0 0 0 0 0 0 1 3 0 1 0 -5 0 0 5 5 0 5 5 5 0 5 5 0 0 0 5 5 5 5 -5
Moose
5 5 0 0 0 0 3 0 0 1 0 5 3 0 5 0 3 3 5 0 0 0 0 0 5 0 0 0 0 0 3 5 0 0 0 0 0 5 -3 0 0 0 5 0 0 0 0 0 0 5 5 0 3 0 0
…
```

The position of the ratings matches the positions of the books in the *booklist.txt* file. For example, the first rating of 5 from Ben applies to *Hitchhiker's Guide to the Galaxy* (`booklist[0]`), and the next 0 means Ben hasn't read *Watership Down* (`booklist[1]`). The meaning of the rating numbers is explained in the table below.

Rating	Meaning
-5	Hated it!
-3	Didn't like it
0	Haven't read it
1	It's okay
3	Liked It
5	Really liked it!

You will determine recommendations for a reader by looking at other readers that are "close" to him or her in their tastes. This is done quite cleverly by computing the **dot product** of their respective ratings, which we will call an "affinity score". A dot product of two *vectors* (i.e., same-size lists) is the sum of the products of their values in corresponding list positions, as explained below. We will call the 2 readers with the highest affinity scores with a given reader the "friends" of that reader.

To illustrate, suppose we had 3 books in our database, and that Rabia rated them [5, 3,-5], Suelyn rated them [1, 5, -3], Bob rated them [5, -3, 5], and Kalid rated them [1, 3, 0]. The affinity between Rabia and Bob is calculated as the following dot product:

$$\begin{pmatrix} 5 \\ 3 \\ -5 \end{pmatrix} \cdot \begin{pmatrix} 5 \\ -3 \\ 5 \end{pmatrix} = (5 \times 5) + (3 \times -3) + (-5 \times 5) = 25 + (-9) + (-25) = -9$$

A **low** or negative score means that the readers are **not similar** in their taste in books. The affinity between Rabia and Suelyn is: (5 x 1) + (3 x 5) + (-5 x -3) = 5 + 15 + 15 = 35. The affinity between Rabia and Kalid is (5 x 1) + (3 x 3) + (-5 x 0) = 5 + 9 + 0 = 14. So Suelyn, having the highest affinity score when compared to Rabia, is more like Rabia in her taste in books than other readers. In general, if both people like a book (rating it with a positive number) it increases their affinity score, and if both people dislike a book (both giving it a negative number), it also increases their affinity score (because a negative multiplied by a negative is positive).

In the sample data above from *ratings.txt*, the affinity score of Ben and Moose is:

$$5 \cdot 5 + 0 \cdot 5 + 0 \cdot 0 + \cdots + 5 \cdot 3 + 5 \cdot 0 + 5 \cdot 0 = 134$$

The higher the dot product of the respective ratings of two people, the more alike are their tastes in books. In this project, you will recommend *all books* that the 2 closest "friends" (the 2 readers with the highest affinity scores when compared to the reader in question) liked (with a rating of 3 or 5) that the reader in question has not yet read (manifest by a rating of 0).

Requirements

When you run your program, you will prompt the user for a reader's name. You will then print the names of the two "friends" and the recommendations for that reader, based on the books those two friends liked. Here is a sample run (user input in boldface):

```
$ python3 bookrecs.py
Enter a reader's name: albus dumbledore
Recommendations for albus dumbledore from joshua and tiffany:
	Douglas Adams, The Hitchhiker's Guide To The Galaxy
	Dan Brown, The Da Vinci Code
	F. Scott Fitzgerald, The Great Gatsby
	Cornelia Funke, Inkheart
	William Goldman, The Princess Bride
	C S Lewis, The Lion the Witch and the Wardrobe
	Gary Paulsen, Hatchet
	Jodi Picoult, My Sister's Keeper
	Philip Pullman, The Golden Compass
	Louis Sachar, Holes
	J R R Tolkien, The Hobbit
	J R R Tolkien, The Lord of the Rings
	Eric Walters, Shattered
	H G Wells, The War Of The Worlds
	John Wyndham, The Chrysalids
$ python3 bookrecs_new.py
Enter a reader's name: chuck
No such reader chuck
```

Note that the 2 "friend" readers appear in **alphabetical order**, and that the list of recommended books are **sorted** by *author last name/author first name/title*. To obtain the result above, you first determine which two readers have the highest "affinity scores" (dot-products of their ratings) compared to *albus dumbledore*. It turns out that those two "friends" are *joshua* and *tiffany*. The recommended books are those that **albus** hasn't yet read that *joshua* or *tiffany* rated a 3 or a 5. Note that to make searching simple, the reader names should be converted to *lower case* as you read them in from the file. Also, the book information is preceded by a tab character ('\t') for indentation.

Your program will also be tested by calling it for various readers, including invalid ones. Print an error message if the reader does not exist, as shown above.

A good way to write this program is to write the following two functions:

- **friends**(name)

  This function returns a person's top 2 friends as a *sorted* list of strings. Sample output:

  friends('megan') ➔ ['roflol', 'shannon']

- **recommend**(name)

  This function returns a *sorted list* of the books recommended for the person, derived from the books liked by the top 2 friends. Remember that the index is the position of the book in the list of books as read in from *booklist.txt*. Sample output:

```
recommend('megan') ➔ [('Meg Cabot', 'The Princess Diaries'), ('Gary
Paulsen', 'Hatchet'), ('Jodi Picoult', "My Sister's Keeper"), ('Jeff
Smith', 'Bone Series')]
```

This list is *sorted* first by author **last** name, then author **first** name, and then by **title**, as mentioned above. Note that the author and title for each book is stored as a **tuple**, *not* a list.

Implementation Notes

There aren't any new Python language features beyond what you learned in CS 1400 required for this project. You may want to review the **sorted** built-in function, which you will need. You will need to use *dictionaries*, *lists*, and *tuples*, as well as file input, along with basic string processing using **split**, **strip**, and **lower**. However, the **operator.itemgetter** method is very useful when finding the top 2 friends, and will be reviewed in class for this module.

Items to consider:

1. When it comes time to recommend books for a reader, *reuven*, say, you need to compute the affinity scores between *reuven* and *all* other readers. (You will do this for *any* reader on demand.) You then need to determine the 2 readers with the highest scores are compared to *reuven.*
2. When `recommend` is called, you call `friends` to get the two similar readers, and then need to determine which books these top 2 friends have rated 3 or 5, collecting only those books that the reader in question hasn't yet read. You need to return a list of pairs containing the author/title of all the recommended books, sorted as explained above. *Don't save this list*, because it could change in a future project where we allow the books, readers and ratings to change. Just return the sorted list as you compute it.

Here is a suggested sequence to develop this project incrementally:

- Read the book data into a **list** of `author,title` *tuples*. Do this at the top level (outside of any function).
- Read the ratings data into a **dictionary** keyed by reader name (*converted to lower case*). The value for each key is a *list* of the ratings for that reader, preserving the original order. Do this also at the top level so the data is available when the functions `friends` and `recommend` are called.
- Write a function `dotprod(x,y)`.
- Write the `friends` function, which returns a sorted list of the names of the two readers with the highest affinity scores compared to the reader in question. Use the **sorted** function to sort those names before returning.
- Write `recommend` by calling `friends` and then getting the recommended books from the two friends obtained. Return the list of books sorted as previously explained.

Make sure each step is complete before moving on to the next. Name your module *bookrecs.py*.

Note: you will need a working version of the **friends** and **recommend** functions described above for a future project! Implement them as specified.

Don't try to do everything at once! Get each piece working before moving on to the next.

FAQs

Q. How do I convert a line of text from *booklist.txt* into a pair for this assignment?
A. `line.strip().split(',')`. (It's good to remove any outer whitespace via **strip**).

Q. How do I read all of a reader's rating scores into a list.
A. Split the line on spaces (the default for **split**).

Q. How do I find the top two friends?
A. This is the interesting part. Suppose you are finding affinity scores for **reuven** compared to all other readers. You first create a *dictionary*, **scores**, say, whose keys are the other readers, and whose respective values are their affinity scores compared to **reuven**:

```
{'ben': 145, 'moose': 60, 'cust1': 39, 'cust2': 78, 'francois': 39, …}
```

You can then sort **scores.items()** by score and choose the first two names. Or you could use **heapq.nlargest**, which uses a **key** function just like **sorted**, but gives you the top **n** entries without sorting the whole thing (just set **n** to 2). Look up **heapq.nlargest** in the Python documentation. Note that you are sorting by the score, which is position 1 of the item.

Q. What if there is a tie for the affinity scores? For example, for *cust1*, *megan* ties for 2nd place with *cust8* with a score of 148. Which one we choose will affect the final recommendations.
A. It doesn't matter. Just use what **nlargest** or **sorted** gives you. Recommendations from friends with equal affinities are equally good.

Q. How do I sort a list of books by last name, then first name, then title?
A. Good question! You write a special **key** function for **sorted** that returns a 3-tuple containing the book data in the desired order.