Assignment 6: Recommender

CompSci 101 Fall 2021

Due: November 30, 2021

Overview

Collaborative filtering and content-based filtering are two kinds of *recommender systems* that provide users with information to help them find and choose anything from books, to movies, to restaurants, to courses based on their own preferences compared to the preferences of others.

In 2009, Netflix held a competition to award one million dollars to a group that could develop a "better" recommender system than the Netflix in-house system. This NY Times Magazine article describes the competition, the winning teams, and how the movie *Napoleon Dynamite* caused problems for the algorithms and ranking/rating systems developed by contest participants. You can watch this short YouTube video about the prize and contest.

In this assignment, you'll develop a program to test different algorithms for recommending items based on the responses made by others. You'll be practicing reading data from files, using Python dictionaries and lists, and sorting data to find good matches. This assignment was adapted from a Nifty Assignment originally developed by Michelle Craig, but modified over several semesters of use at Duke in CompSci 101.

Completing the Assignment

You'll need to use the starter code from this zip file and create a project from it. The zip file can be found here. Download the zip file, unzip the file, then go into Pycharm and select *File* > *Open…* then select the unzipped folder. Choose either New Window or This Window.

Due Date

This assignment is due on **Tuesday, November 30** at 11:59pm, and its grace period will go until **Friday, December 3 at 11:59pm** (so it lines up with prior assignment grace periods). There is ***no late period*** after that. No assignments/submissions will be accepted after **Friday, December 3 at 11:59pm.**

Note we are giving the assignment out early, you have over 2 weeks to complete it. Start early!

Requirements and Modules to complete

You must complete several modules described below.

1. ***RecommenderEngine.py*** -- This module is the basis for making recommendations from data related to ratings of individual items by raters/users. You must complete this module to make recommendations in different areas: movies, books, food, and courses.
2. ***TestRecommender.py*** -- Using SmallDukeEatsReader.py and the examples given in this write-up, you should write code to test the key functions in RecommenderEngine.py. Calling **`SmallDukeEatsReader.getdata()`** will read example data and return it for you to use. You're given some code to start with, but you must also add tests to the TestRecommender module.
3. ***X-Reader*** modules -- These modules should read data files in different formats and return a list of items and a dictionary of ratings corresponding to these items in the format outlined below. You are required to write ***MovieReader.py*** and ***BookReader.py***.
4. ***RecommenderMaker.py*** -- When complete, this module should make recommendations for movies or books or courses or eateries.

Completing RecommenderEngine.py

You'll need to complete three functions that will be called by the modules you write to make specific recommendations about movies, books, eateries, courses, etc. You'll likely want to complete helper functions that these three functions call. Creating helper functions will make it easier for you to develop and test the functions described here.

We strongly suggest that you write the functions in this order. This means you should be sure to write, test, debug, and submit to Gradescope for each one before starting the next one.

1. The function **`averages`**
2. The function **`similarities`**
3. The function **`recommendations`**

These functions have three general kinds of parameters:

- **`items`**, a list of strings
- **`ratings`**, a dictionary of raters/users to a list of integer ratings
- **`name`**, the name of a rater/user that's a key in the dictionary **`ratings`**

Here is an example of strings that you'll get in the file **`eateries.txt`** as a list of items:

```
items = ["DivinityCafe", "FarmStead", "IlForno",
   "LoopPizzaGrill", "McDonalds", "PandaExpress",
    "Tandoor", "TheCommons", "TheSkillet"]
```

For these eateries, here's a dictionary of raters/users to ratings. This is stored in the file **`food.json`** that you get for this assignment.

```
ratings = {"Sarah Lee":
             [3, 3, 3, 3, 0, -3, 5, 0, -3],
           "Melanie":
             [5, 0, 3, 0, 1, 3, 3, 3, 1],
            "J J":
             [0, 1, 0, -1, 1, 1, 3, 0, 1],
            "Sly one":
             [5, 0, 1, 3, 0, 0, 3, 3, 3],
            "Sung-Hoon":
             [0, -1, -1, 5, 1, 3, -3, 1, -3],
            "Nana Grace":
             [5, 0, 3, -5, -1, 0, 1, 3, 0],
            "Harry":
             [5, 3, 0, -1, -3, -5, 0, 5, 1],
            "Wei":
             [1, 1, 0, 3, -1, 0, 5, 3, 0]}
```

Note that each value in the dictionary has the same number of entries as the list of items. In general, a value in the dictionary is one user's list of ratings, and **`value[k]`** is the rating given to **`item[k]`** (a.k.a. parallel lists). So for this example, **`ratings["Melanie"][2]`** is 3, meaning that 3 is the rating Melanie gives "IlForno", the eatery in **`items[2]`**.

Functions to Implement

This module has three functions to implement. You can test these functions with TestRecommender.py.

`averages(items, ratings)`

This function calculates the average ratings for items. In calculating averages, you should not count raters who give a value of 0 (meaning "not rated"). In other words, do not include them in the count for the denominator when calculating the averages. If there are no ratings for an item, then it should have an average rating of 0.

Parameters:

- items (type: list of strings) - list containing the names of the items to be rated

- ratings (type: dictionary) - a dictionary where the key is the rater's name and the values are their corresponding ratings.

Returns:
Type: A list of tuples, where the first element is a string and the second element is a float.

A list of tuples is returned, where the first element of each tuple is a string representing the item being rated, the second element is a float representing the average rating for that item. The list should be **sorted so that the highest rated item is first**, and **ties should be sorted alphabetically**.

Example Return Value:

```
[('item1', 13.5), ('item2', 4.6), … ]
```

`similarities(name, ratings)`

This function calculates how similar the rater called **`name`** is to each of the other raters, based on the ratings they've given to items. Conceptually, if two users both give either a negative or positive rating to some item, that makes these users more "similar" than if one had given a positive rating and the other had given a negative rating. We can compute a similarity measure between two raters that follows this idea by finding the dot-product of their rating-lists.

The dot-product is related to a measure of the "angle" between two ratings in a mathematical ratings space. To find the dot-product of two lists, go down both lists one index at a time and multiply the rating values that occupy the same indices, and then add these products together.

For example, we can compute the dot-product for the rating lists [-3,0,5,3] and [-1,3,0,5] by multiplying the corresponding elements of the lists and then summing up these products. This yields a dot-product ("similarity score") of (-3)*(-1) + (0)*(3) + (5)*(0) + (3)*(5) = 3 + 15 = 18.

For the lists [-3,0,5,3] and [3,0,-3,3], the dot-product is (-3)*(3) + (0)*(0) + (5)*(-3) + (3)*(3) = -9 + (-15) + 9 = -15. Thus, when being compared to the rater with ratings list [-3,0,5,3], the rater with list [-1,3,0,5] is more similar than the rater with list [3,0,-3,3], since their "similarity scores" are 18 and -15, respectively.

Note: The rater whose name is **`name`** should not be evaluated as to how similar they are to themself. In other words, do not compute a similarity score for **`name`** with respect to themself.

Parameters:
- name (type: string) - name of the rater in which to find similarities to

- ratings (type: dictionary) - a dictionary where the key is the rater's name and the values are their corresponding ratings.

Returns:
Type: A list of tuples, where the first element of each tuple is a string and the second element is an integer.

This function returns a list of tuples where the first element is a rater's name (string) and the second element is a "similarity score" (int). This represents the similarities between each of the other raters in the ratings dictionary and the rater specified by the parameter **`name`**. The list is **sorted with the highest "similarity score" first**, and **ties should be sorted alphabetically**.

The list returned ***should have one less element than the number of (key, value) pairs in the ratings*** since there will be no entry corresponding to the similarity of **`name`** with themself.

Example Return Value:

```
[('Chris', 15), ('Sam', 10), … ]
```

`recommendations(name, items, ratings, numUsers)`

This function calculates weighted average ratings of items and makes recommendations based on the parameters and these weighted averages. ***You must call*** **`averages`** ***and*** **`similarities`** ***when implementing this function.***

You will need to call **`similarities`** and use the results of the returned list to calculate a weighted average using **`averages`** as described below. The idea is to weight the ratings of raters who are similar to **`name`** more than the ratings of those with whom **`name`** doesn't agree. Consider an example with the following ratings:

```
ratings = {"me":
             [5, 3,-5],
           "rater1":
             [1, 5,-3],
           "rater2":
             [5,-3, 5],
           "rater3":
             [1, 3, 0]}
```

The similarity measures (e.g., as returned by **`similarities`**) with respect to the rater named **`me`** (i.e. **`name = "me"`**) are computed as:

```
1*5 +  5*3 + -3*-5 = 35
5*5 + -3*3 +  5*-5 = -9
1*5 +  3*3 +  0*-5 = 14
```

Thus, **`similarities`** returns the following list (sorted by similarity score):

```
[('rater1', 35), ('rater3', 14), ('rater2', -9)]
```

So when computing recommendations for **`me`**, **`rater1`**'s ratings should be weighted by 35, **`rater2`**'s ratings should be weighted by -9, and **`rater3`**'s ratings should be weighted by 14. We apply these "weights" to create new weighted ratings as follows:

```
35 * [1, 5,-3] = [ 35, 175,-105]
-9 * [5,-3, 5] = [-45,  27, -45]
14 * [1, 3, 0] = [ 14,  42,   0]
```

You can store these weighted ratings in a new dictionary, perhaps called **`newRatings`**, and then calculate new "weighted average" ratings for the items by calling **`averages`**. In this case, the sum of the ratings (down the columns) is **`[4, 244, -150]`**, and so **`averages`** would return **`[4/3, 244/3, -150/2] = [1.33, 81.33, -75]`**.

Remember, **`averages`** should divide each item's total rating by the number of ratings that were taken into account (i.e. the number of nonzero ratings). All three raters have rated the first two items, so the totals were divided by 3. For the last item, since **`rater3`** did not rate it (i.e. their rating for the item is 0), we only took two ratings into account, so the total was divided by 2.

Finally, we conclude that for the user named **`me`**, the most highly-recommended item is the item whose score is 81.33, the next most highly-recommended is the item whose score is 1.33, and the least-recommended is the item whose score is -75.

The parameter **`numUsers`** is an integer denoting the number of users (besides **`name`**) who should be used in computing weighted averages. ***Only use weighted ratings for the first*** **`numUsers`** ***raters*** in the list returned by **`similarities`**. Since the list returned by **`similarities`** is sorted, you'll be weighting ratings by the **`numUsers`** closest (most "similar") users to the rater whose name is **`name`**.

Parameters:

- name (type: string) - name of the rater that recommendations will be based upon
- items (type: list of strings) - list containing the names of the items to be rated
- ratings (type: dictionary) - a dictionary where the key is a rater's name and the values are their corresponding ratings.
- numUsers (type: integer) - the number of users whose ratings will be part of the collaborative filter process to find recommendations.

Returns:
Type: A list of tuples, where each tuple has a string as its first element, a float as its second element.

This function returns a list of tuples where the first element is the name of an item (string) and the second element is ***the weighted average*** (float) for that item as calculated using the method described above. The list is sorted with the highest recommended item first, ties should be sorted alphabetically.

Example Return Value:

```
[('item1', 3.5), ('item2', 1.0), … ]
```

X-Reader Modules

The MovieReader.py Module

This module has one function you must write. When reading **`movies.txt`**, only go through the file once and store the contents in a list; ***do not*** use functions like **`f.seek(0)`** to go through the file multiple times.

`getdata()`

The file **`data/movies.txt`** contains 22,930 lines in the format shown below (these are the first three and last three lines):

```
student1367,Star Trek Beyond,3
student1367,Rogue One,3
student1367,Moana,1
…
student1460,Pirates of the Caribbean: On Stranger Tides,1
student1460,The Dark Knight,5
student1460,Avatar,5
```

In general, each line of the file has the following format: **`studentID,Movie,Rating`**
You must write **`MovieReader.getdata()`** to read this data file **`data/movies.txt`** and return the appropriate values described below.

Parameters:
No parameters

Returns:

Type: A two-tuple. The first element is a list of strings and the second element is a dictionary.

The first element is the list of movies being rated (***items***), and the second element is a dictionary of raters to ratings where the key is a rater name, e.g. “student1001” and the value is a list of integers (one for each movie), with a zero/0 given for an unrated movie (***ratings***).

The list of movies returned should be in alphabetical order. For every key in the dictionary ratings, the numerical rating found in **`ratings[key][dex]`** is the rating for the movie whose name is in **`items[dex]`** (a.k.a. a parallel list). In other words, the list found at **`ratings[key]`** should be ordered in parallel with the list **`items`**. That's an invariant for the relationship between the returned values items and ratings -- and holds for every list that's a value in **`ratings`**.

The BookReader.py Module

This module has one function you must write. When reading **`books.txt`**, only go through the file once and store the contents in a list; ***do not*** use functions like **`f.seek(0)`** to go through the file multiple times.

`getdata()`

The file **`data/books.txt`** has ratings in a different format from that found in **`movies.txt`**.
Part of the first two and last three lines of **`books.txt`** are reproduced below:

```
student1001,The Hitchhiker's Guide To The Galaxy,5,Watership Down,0,...
student1002,The Hitchhiker's Guide To The Galaxy,5,Watership Down,5,...
...
student1085,The Hitchhiker's Guide To The Galaxy,0,Watership Down,0,...
student1086,The Hitchhiker's Guide To The Galaxy,5,Watership Down,5,...
student1087,The Hitchhiker's Guide To The Galaxy,3,Watership Down,5,...
```

Every line in **`books.txt`** has the same titles in the same order. The general format for each line is **`rater,title,rating,title,rating,...`** You can see the file for more details.

You must write **`BookReader.getdata()`** to read this data file **`data/books.txt`** and return the appropriate values described below.

Parameters:
No parameters

Returns:
Type: A two-tuple. The first element is a list of strings and the second element is a dictionary.

The first element is a list of the book titles rated. The second element is **`ratings`**, a dictionary of user/raters to lists of ratings. The same invariant holds for these that held in the ***MovieReader*** module: the book title in **`items[k]`** is rated in **`ratings[name][k]`** for every name in the data file **`books.txt`** (a.k.a. a parallel list). The order of the titles in the returned list **`items`** should be the same as that in which the titles appear on every line of **`books.txt`**.

RecommenderMaker Module

This module has one function you must write. Testing the **`makerecs`** function in the main block of RecommenderMaker is optional, but recommended.

`makerecs(name, items, ratings, numUsers, top)`

This function should organize the recommended items that are returned by the function **`RecommenderEngine.recommendations`**. It should return two lists that both have length **`top`**. In other words, the parameter **`top`** is an integer denoting the number of "top" results that should be included in each returned list. Each list thus contains a portion of the "best" tuples returned when calling **`recommendations`** with the first four parameters passed to **`makerecs`**.

The first list returned contains items rated by the user whose name is in the first parameter to **`makerecs`**. The second list contains items that are especially "recommended" for that user, since these haven't been rated by them. You'll need to write code to determine the top rated and unrated items in the list returned from **`RecommenderEngine.recommendations`**. Remember to slice the two lists before they are returned by **`makerecs`** so that they both have length **`top`**.

You are encouraged to test the **`makerecs`** function in the main block of RecommenderMaker.py. Using the example given below, you can write code to print the top 3 movies for **`'student1367'`** that they've seen and the top 3 that they have not seen (which are thus recommended for them). You can test **`makerecs`** using different sets of **`items`** and **`ratings`** as well, such as those returned by MovieReader and BookReader.

Parameters:

- name (type: string) - name of the rater that recommendations will be based upon
- items (type: list of strings) - list containing the names of the items to be rated
- ratings (type: dictionary) - a dictionary where the key is a rater's name and the values are their corresponding ratings.
- numUsers (type: integer) - the number of users besides **`name`** whose ratings will be used to calculate the weighted average
- top (type: integer) - the number of items to include

Returns:

Type: A two-tuple consisting of two lists:

- The first list is the top items rated by name (string)
- The second list is the top items not seen/rated by name (string)

Example:
Consider passing the following parameters to **`makerecs`**:

```
name = 'student1367'
items = ['127 Hours', 'The Godfather', '50 First Dates', 'A Beautiful
Mind', 'A Nightmare on Elm Street', 'Alice in Wonderland', 'Anchorman:
The Legend of Ron Burgundy', 'Austin Powers in Goldmember', 'Avatar',
'Black Swan']

ratings = {'student1367': [ 0, 3,-5, 0, 0, 1, 5, 1, 3, 0],
           'student1046': [ 0, 0, 0, 3, 0, 0, 0, 0, 3, 5],
           'student1206': [-5, 0, 1, 0, 3, 0, 5, 3, 3, 0],
           'student1103': [-3, 3,-3, 5, 0, 0, 5, 3, 5, 5]}

numUsers = 2
top = 3
```

The function call in the main block of **`RecommenderMaker`** would be as follows:

```
makerecs(name, items, ratings, numUsers, top)
```

First, **`makerecs`** calls **`RecommenderEngine.recommendations`**. The similarities calculated in the call to **`RecommenderEngine.similarities`** made by **`RecommenderEngine.recommendations`** returns the following list of tuples:

```
[('student1103', 67), ('student1206', 32), ('student1046', 9)]
```

`RecommenderEngine.recommendations` will then keep the top **`numUsers`** tuples that are most similar to **`name`**. Thus, since **`numUsers`** is 2 in this example, the following tuples are kept:

```
[('student1103', 67), ('student1206', 32)]
```

Using these values, **`RecommenderEngine.recommendations`** calculates the weighted ratings (**`newRatings`**) to be:

```
{'student1103': [-201, 201, -201, 335, 0, 0, 335, 201, 335, 335],
 'student1206': [-160, 0, 32, 0, 96, 0, 160, 96, 96, 0]}
```

Thus, the list weighted averages returned by **`RecommenderEngine.recommendations`** after calling **`RecommenderEngine.averages`** is:

```
[('A Beautiful Mind', 335.0), ('Black Swan', 335.0), ('Anchorman: The
Legend of Ron Burgundy', 247.5), ('Avatar', 215.5), ('The Godfather',
201.0), ('Austin Powers in Goldmember', 148.5), ('A Nightmare on Elm
Street', 96.0), ('Alice in Wonderland', 0.0), ('50 First Dates',
-84.5), ('127 Hours', -180.5)]
```

Finally, ***the desired return value*** of **`makerecs`** is a tuple consisting of two lists, both with length **`top`** (3 in this example). The first list contains tuples corresponding to the items which **`name`** (here, **`'student1367'`**) has already rated/experienced (based on if the corresponding entries in the ratings dictionary are 0 or not):

```
[('Anchorman: The Legend of Ron Burgundy', 247.5),
 ('Avatar', 215.5), ('The Godfather', 201.0)]
```

The second list contains tuples corresponding to items which **`name`** has not already rated:

```
[('A Beautiful Mind', 335.0), ('Black Swan', 335.0),
 ('A Nightmare on Elm Street', 96.0)]
```

Testing

Required

You must use the provided ***TestRecommender.py*** module for testing your three functions from RecommenderEngine. You will verify that these functions work by modifying and running TestRecommender to call these functions using data from the provided SmallDukeEatsReader module.

Using the provided testing code in TestRecommender allows you to test your functions using the data returned by SmallDukeEatsReader.py. However, this code won't work until you've implemented the functions in the RecommenderEngine module.

The output for the function **`averages`** from **`RecommenderEngine`** is a list of tuples sorted by average rating. The number of decimal places doesn't matter when printed. When checking if two floats are mostly equal, a common method is subtracting the values and checking if the absolute value is less than a threshold (i.e. **`abs(a-b) < 0.001`**).

```
[('DivinityCafe', 4.0), ('TheCommons', 3.0),
 ('Tandoor', 2.4285714285714284), ('IlForno', 1.8),
 ('FarmStead', 1.4), ('LoopPizzaGrill', 1.0),
```

```
 ('TheSkillet', 0.0), ('PandaExpress', -0.2),
 ('McDonalds', -0.3333333333333333)]
```

The output for **`similarities`** for the user Sung-Hoon is:

```
[('Wei', 1), ('Sly one', -1), ('Melanie', -2), ('Sarah Lee', -6),
 ('J J', -14), ('Harry', -24), ('Nana Grace', -29)]
```

Recommendations for Sarah Lee using the three (3) closest users are:

```
[('Tandoor', 149.5), ('TheCommons', 128.0),
 ('DivinityCafe', 123.33333333333333), ('FarmStead', 69.5),
 ('TheSkillet', 66.0), ('LoopPizzaGrill', 62.0),
 ('IlForno', 33.0), ('McDonalds', -69.5),
 ('PandaExpress', -165.0)]
```

You must modify TestRecommender so that instead of just printing results that you must examine by "eyeball", ***the code runs and prints verification that the functions work*** (for example, "averages works", or "averages fails") by comparing the results that are actually returned to what should be returned. This should be done for all three functions from RecommenderEngine. You can use the example outputs shown above in your comparisons.

Optional Testing

Once the functions in ***MovieReader*** and ***BookReader*** have been implemented to return the list of items and the ratings for these items, you may want to test these as follows. You could, for example, create smaller text files in the same format as **`books.txt`** and **`movies.txt`**, perhaps using the data from **`eateries.txt`** and **`food.json`** that the TestRecommender module uses. Then you'll be able to judge the correctness of the return values in MovieReader and BookReader on small files instead of the larger data files you're given.

When testing the functions in RecommenderEngine, you can optionally use MovieReader.getdata() and BookReader.getdata() in addition to SmallDukeEatsReader. When you’re convinced that the functions in RecommenderEngine work, ideally you'll create code to test the function in ***RecommenderMaker*** as well by similarly comparing its return value to what you would expect.

Submitting and Grading

1. Log in to Gradescope.
2. Under the CompSci 101 dashboard, click Assignment 6: Recommender.

3. Click Submit Programming Assignment, click “Click to browse”, and select the required files. Click “Browse Files” again to select and submit more files.
4. Submit every .py module you're asked to write/modify. This includes:
    a. RecommenderEngine
    b. TestRecommender
    c. MovieReader
    d. BookReader
    e. RecommenderMaker
5. Once all files are selected, click Upload.

Autograded portion [49 points]

- [20] X-Reader Modules
    - [10] MovieReader.getdata()
        - Returns a tuple consisting of a list and a dictionary
        - List contains correct values
        - List is sorted correctly
        - Dictionary contains correct keys and values
    - [10] BookReader.getdata()
        - Returns a tuple consisting of a list and a dictionary
        - List contains correct values
        - List is sorted correctly
        - Dictionary contains correct keys and values
- [18] RecommenderEngine
    - [5] averages()
        - Returns a list of tuples
        - Output gives correct values
        - Output is sorted correctly
    - [5] similarities()
        - Returns a list of tuples
        - Output gives correct values
        - Output is sorted correctly
    - [8] recommendations()
        - Returns a list tuples
        - Output gives correct values
        - Output is sorted correctly
- [10] RecommenderMaker
    - [10] makerecs()
        - Returns a tuple of lists
        - First element of tuple is a list containing correct values
        - Second element of tuple is a list containing correct values
- [1] TestRecommender
    - [1] File is submitted

Hand-graded portion [6 points]

- [3] TestRecommender
    - [1] Compares the return value of averages() with a reference value and prints a message based on correctness
    - [1] Compares the return value of similarities() with a reference value and prints a message based on correctness
    - [1] Compares the return value of recommendations() with a reference value and prints a message based on correctness
- [1] RecommenderEngine
    - [1] recommendations() calls averages() and similarities()
- [2] Rest of submission
    - [2] Name and docstring comments (including for any additional helper functions)
        - All required modules should have your name: RecommenderEngine, RecommenderMaker, MovieReader, BookReader, and TestRecommender
        - Docstring comments are required for functions in RecommenderEngine, RecommenderMaker, MovieReader, and BookReader