CISC-121 Winter 2013

Weeks 5 and 6 Lab Assignment: Making Recommendations

This is the assignment for the labs in Weeks 5 and 6 (February 4 – 15). **This assignment will be graded, and is worth 5% of the course. See below for details of how to submit your solution.**

Deadline:

Please complete your solution by 8:00 AM Monday February 18. I will give you a 24-hour grace period that ends at 8:00 AM Tuesday February 19. Assignments submitted during the grace period will be penalized 10% for lateness. Assignments will not be accepted after the end of the grace period.

Submitting Your Solution:

You are permitted (but not required) to work in pairs on this assignment. Your submission should be in the form of a single file containing your Python program for this lab. At the top of the submission you **MUST** include the following information:

- **full name** and **student number** of the person(s) who worked on this solution.
- the date on which you are submitting.

If you work with another person, **only one of you** should submit the solution. It does not matter which of you submits.

Submit your solution using the CISC-121 Moodle page. This assignment is listed as an activity for the sixth week of the course.

Introduction:

These days it seems like every website you visit, from Amazon to Flickr to twitter to Netflix, wants to recommend something to you – books you might like to read, people you might like to meet, photographs you might like to look at, games you might enjoy playing. Recommendation systems have become an integral aspect of the World Wide Web.

So how do they work? Pretty much the way you might guess: they gather information about lots of customers, including you, and try to find people with similar tastes. In the case of a site like Amazon or Netflix, they try to recommend a product to you that "people like you"

have rated highly.

In this assignment you will write a very simple recommendation system for customers of an ice cream vendor. The company currently has 38 customers and 32 flavours of ice cream. (If you have the urge you can add more customers, including yourself, and more flavours.) Each customer has already tried some flavours, and has rated each flavour they have tasted on a scale of 1 to 10, where 1 means "hated it" and 10 means "loved it".

In the Python code provided, these ratings are randomly generated and stored in a nested dictionary called customer_ratings. Each element of customer_ratings has a customer name as its key, and the data value is another dictionary (hence the term "nested dictionary"). The embedded dictionary for each customer has one entry for each flavour that the customer has rated, consisting of the flavour name as key and the integer rating as the value. For example, if there were only three customers and a few flavours, the customer_ratings dictionary might look like this:

```
customer_ratings = { "Hudson": {"vanilla": 5 , "chocolate":8},
                     "Simon" : {"chocolate": 1, "strawberry ripple": 6,
                                "rocky road":1},
                     "Rally" : {"strawberry ripple":9, "chocolate": 2}
                   }
```

We can access the data in a nested dictionary using some of the standard Python techniques that work for lists and other iterable objects. For example,

```
for c in customer_ratings:
      for f in customer_ratings[c]:
            print c, f, customer_ratings[c][f]
```

will print all the individual ratings. Try this on a small example to make sure you are comfortable with accessing the information. Remember that searching a dictionary for an element that is not present causes an error. To avoid this, the **"in"** test is our best friend.

For example

```
p = enterbox("Enter a customer's name")
print customer_ratings[p]
```

will cause an error if there is no such person in the dictionary, but

```
p = enterbox("Enter a customer's name")
if p in customer_ratings:
      print customer_ratings[p]
```

```
else:
      print “Unknown customer”
```

works just fine, and

```
p = enterbox(“Enter a customer's name”)
while p not in customer_ratings:
      p = enterbox(p + “ is unknown.  Enter a customer's name”)
print customer_ratings[p]
```

is even better. Or you could use the easygui function called **choicebox** to limit the user's choice to existing customers.

The next question is how to measure the similarity of two customers. This is a difficult and slightly vague question, so as you might imagine, many different ideas have been proposed. It's not hard to come up with a mathematical definition of similarity – the problem is finding one that matches our intuitive understanding of what it means for two people (or other objects) to be similar. For example, suppose Albert and Kurt have both rated chocolate and vanilla, and no other flavours, and their ratings are

“Albert”:{“chocolate”:8, “vanilla”:4}
“Kurt” :{“chocolate”:2,”vanilla”:1}

You could say that Albert and Kurt are very different, because Albert likes each flavour four times as much as Kurt does. But you could also say they are very similar, because they each like chocolate twice as much as they like vanilla.

Some widely used measures of similarity are quite complex to compute. I encourage you to research this on your own. In this lab we will use a simple measure of similarity which comes from **fuzzy set theory** (which you will see again in upper year CISC courses).

Suppose we want to compute the similarity of two customers, p1 and p2.

- First we compute the sum of all p1's ratings, and the sum of all p2's ratings. Call these p1_sum and p2_sum.
- Next we compute the intersection of p1's ratings and p2's ratings: for each flavour they have both rated, we choose the **minimum** of the ratings the two people gave that flavour (don't worry, there's an example below) and we add up those minimum ratings. Call this intersection_sum.
- We compute intersection_sum/p1_sum, and intersection_sum/p2_sum
- The smaller of these two values is the similarity of p1 and p2

Example: Suppose Bugs (p1) and Elmer (p2) have rated the following flavours:

“Bugs” : {“lemon”:6, “maple walnut”:2, “chocolate”: 5, “bubble gum”: 4}
“Elmer” : {“mocha”: 4, “lemon”: 5, “bubble gum”: 7}

p1_sum = 6+2+5+4 = 17 (the sum of Bugs's ratings)
p2_sum = 4+5+7 = 16 (the sum of Elmer's ratings)
intersection_sum = min(6,5) + min(4,7) (since they both rated lemon and bubble gum)
= 5+4 = 9
similarity = min (9/17, 9/16) (remembering to make sure these are not both evaluated to 0 due to integer division)
= 9/17

You can see that if p1 and p2 have exactly the same ratings on exactly the same flavours, their similarity will be 1. If they have not rated any of the same flavours, their similarity will be 0. For all other situations, their similarity will be somewhere between 0 and 1.

Another (less sophisticated) method for recommendation is to recommend the flavour that has the highest average rating among all the flavours that the person has not yet rated themselves. For example, if Daffy has not yet rated "chocolate" and "strawberry", and "chocolate" has an average rating of 5.3 and "strawberry" has an average rating of 6.2, then this method would recommend "strawberry" for Daffy to try.

The Assignment (finally!):

1. Download and familiarize yourself with the Python code provided. It creates the customer list, the flavour list, and the dictionary of customer ratings. You do not need to modify this code, but you may wish to add a comment at the beginning to describe your program, and you may wish to insert the functions you will be defining above the creation of the data lists. In case you are wondering, the flavours are all real ice cream flavours from various parts of the world. (The customer names are all drawn from a particular genre of film and literature. A total of 0 Bonus Marks will be awarded to anyone who identifies the source for each name without using an internet search engine.)

2. Create a function with a name like **fuzzy_similarity**, which takes the dictionary of customer ratings and two customer names as arguments, and returns the similarity value for the two customers.

3. Create a function with a name like **best_match**, which takes the dictionary of customer ratings and a single customer name as arguments, and returns the name of the customer who has the highest similarity with the given customer (of course it should

not return the customer's own name!)

4. Create a function with a name like **recommend,** which takes the dictionary of customer ratings and a single customer name as arguments, and returns a recommendation of a new flavour for the given customer as follows:
   1. find another customer who is the most similar to the given customer
   2. From the flavours the other customer has rated that the given customer has not, return the flavour that the other customer rated the highest. If the given customer has already rated all the flavours rated by the other customer, return a string like "No recommendation" (or, and this is **completely optional**, go to town on the problem and try again with the customer who is the second-best match for the given customer, and so on).

5. Another method for recommendation is to recommend the flavour that has the highest average rating among all the flavours that the person has not yet rated themselves. For example, if "chocolate" and "strawberry" are the only flavours that Daffy has not rated, and "chocolate" has an average rating of 5.3 and "strawberry" has an average rating of 6.2, then this method would recommend "strawberry" for Daffy to try. Write a function with a name like **other_recommend**, which takes a dictionary of customer ratings and a single customer name as arguments, and returns a recommendation based on the method just described. As in Part 4, if the customer has already rated all flavours your function should return "No recommendation" or something similar.

6. Write a main program that prompts the user for a customer name and then displays the recommendations for that customer, if they can be made, using the two functions you created in Part 4 and Part 5.

7. Add a comment to the end of your program in which you answer the following questions: Do the two recommendation methods give consistent results? If not, which method do you think gives better recommendations?

Dealing with Errors:

Your program and the functions you write should be robust – that is, the main program should check user input, and the functions should check their arguments to prevent any error messages. For example, your functions should not try to access a person's ratings without first checking that the person is in the dictionary. If these safety checks fail, your program/function should take appropriate action or return a flag value such as None. When you call a function that might return None, you should check the returned value before

continuing.

Documentation and Programming Style:

Each function and each major block of code must have a comment describing its purpose. Further comments should be added as you feel appropriate to clarify the operation of your program.

Your program should exhibit good programming structure (appropriate use of functions, no reference to global variables, etc.) and style (visual layout, variable names, etc.)

Testing:

As with our previous labs, exhaustive testing on even a moderate amount of data is infeasible. To validate your code you might try reducing the number of customers to 3 and the number of flavours to 3. If you print the entire customer_ratings dictionary you can check the recommendations by hand.

Marking Scheme:

This assignment will be marked out of 100:

1. PART 2 (fuzzy similarity): 10
2. PART 3 (best match): 10
3. PART 4 (recommend): 10
4. PART 5 (other_recommend): 10
5. PART 6 (main program): 10
6. PART 7 (answer questions): 10
7. DEALING WITH ERRORS : 10
8. COMMENTS : 15
9. PROGRAMMING STYLE : 15

The Rest of the Story:

This assignment is a very shallow dive into the field of data mining: looking for patterns in a body of information, and making predictions or drawing conclusions based on the patterns found.

The algorithm for making a recommendation (Step 4) is very simple. There are much more sophisticated algorithms available, and we may return to this question in a future lab. For those who are interested in this problem, I highly recommend the book "Programming Collective Intelligence" by Toby Segaran. This is an excellent resource book with all code samples written in Python.