Reciprocal Recommendation
Plus introduction to Python

TI2716-C Multimedia Analysis

2018-2019 Q3 Week 1
14 February 2019

How to submit your work:

In the Lab 1 Google Docs folder:
https://drive.google.com/drive/folders/1L3lHxFUKgkkig3D3yjqREyuDxR-nOD-t?usp=sharing

create a doc file and name it `firstname_last name`. Here `firstname` should be your *roepnaam*.

This doc file is your lab report. Share it for editing with the course instructors: hayleyhung@gmail.com, martha.larson@gmail.com, elnouty.amira@gmail.com ,mariswel@hotmail.com, shivam2239@gmail.com, kapcakoyku@gmail.com, stephanietan0514@gmail.com, josedvq@gmail.com, ekn.gedik@gmail.com, (nB. Please use TU Delft addresses to contact the teaching team. For example, use m.a.larson@tudelft.nl and h.hung@tudelft.nl to contact Martha Larson and Hayley Hung, and not these gmail addresses.)

Every time you see a Google Docs logo next to the title of a section of this lab, you should include the requested output and the responses to the questions in that section in your lab report.

Our goals today:

Your overall mission is to build a simple recommender system. The recommender system is a "pairing algorithm", which will pair people in this course based on how they rate apps. For this task we will use the set of ratings that we previously collected.

On your way to accomplishing this mission, you will be introduced to Python, since a basic mastery of Python is a key skill in this course. The readability and extensibility of Python will allow you to gain hands-on experience with analyzing multimedia data, and as a result understanding of multimedia analysis algorithms and principles. For many, Python skills will come in handy in the future.

During the course of this lab, you will also have a chance to compare different similarity measures (Tuesday's lecture) that can be used in collaborative filtering algorithms (Friday's lecture). We will also see how we can have different pairings by using different similarity functions.

At the end of today, you will have acquired/practiced the following aspects of Python:

- Running a Python program from the command line.
- Printing to screen
- Lists
- Tuples
- Functions
- Modules
- Loading files
- Dictionaries

The very basics of Python will be covered here, and we will not go in much depth. If you know Python already and think you can implement the exercises in a better way, we encourage you to do so. **If you're really already familiar with Python, skip to 1.1.4, where you start to code independently**.

If you want to use different tools during this lab, for example another editor, you are welcome to do so. However, please assume that the student assistants will not be able to answer your questions about tools and environments other than those distributed with the course VM. The use of different programming languages, like Java or C++, is explicitly discouraged since we will be using tools that are based on Python in the forthcoming labs.

1.1 Introduction to Python

Python is an interpreted language which has recently gained much popularity. This is due to its obvious simplicity (forget the heavy grammar and strict data types of c++ and Java), great extensibility (so called modules, or "libraries"), and friendliness when dealing with different kind of data. All glue together with a great and fast performance.

During this course, we will be using **Python 2.7**

Some general background about Python:
Python has two branches, 2.x and 3.x, which are being developed in parallel and each of them has different features. Watch out as they are not so easily interchangeable! However you don't need to worry about it in this course. The most widely-used version is still 2.7, as it has the most number of modules available. However the 3.x has a few better improvements, especially in the area of string decoding (all strings are UNICODE).

1.1.1) How to use the Terminal?

Opening the terminal can be done in a few ways. Some of them are:
- Left-click on the icon in the bottom left corner of the screen
- Right-click to open the context menu in the folder you want to use the Terminal, and select 'Open in Terminal'
- Pressing Ctrl-Alt-T

The most basic commands for navigating the terminal are:

`ls`	(lists files and folders)
`cd FolderName`	(navigates to the named folder.)
`cd ..`	(Moves up one level in the file system)
`tab`	(autocomplete filename or command)
`mkdir FolderName`	(create a directory named Foldername)

1.1.2) Creating and running a Hello World Python program
Using the terminal, navigate to `/home/student/MMA3/` in the VM and create a folder named `Exercise 1` using the command `mkdir Exercise\ 1` (Note: the "\ " is a space in the terminal. Now enter the folder.
Create a new file named `hello.py` this can be done for example with the text editor gedit, by using the command `gedit hello.py`
In the opened file type:

```
      print('Hello World!')
```

Save the file and close gedit. In the terminal type

```
      python hello.py
```

You've now created and run a Python program. You'll see your text printed into the console.

1.1.3) Using geany and print
Going back and forth between the Terminal and the gedit is bit cumbersome, so let's just use an Integrated Development Environment (IDE). Open up geany, which can be found on the desktop, or you can just type `geany` into the terminal you already have open.

Create the file `pair.py` using geany, and save it in the Exercise 1 folder. On the first line add

```
      print("Exercise 1 - Pairer")
```

Note that we're using double quotes that are different than the single quotes that we used earlier in the Hello World program. Python doesn't care which ones you use to delineate a string. Press F5 to run the program. Press Enter to close the terminal that pops up during execution of your program.

1.1.4) Printing, lists and loops
Declare a list and show its contents as follows:

```
      apps = ['Meaning of Life', 'Hollywood Bowl', 'Life of Brian', 'Holy Grail', 
'Something Completely Different']
      print(apps)
```

We can print each movie on a separate line by iterating through them. Instead of `print(apps)` use:

```
      for m in apps:
            print m
```

The print statement is a funny thing in Python 2, it is a statement and not a function. That's why it also works without (). (FYI: In Python 3 it is a function.)

Note the lack of curly brackets ({,} like in C-style languages) or end-statements (Matlab, etc). Instead, Python uses indentation to indicate sections of code. Be aware, ***Lack of indentation causes indentation errors***. This is one of the most common errors when you are starting with Python.
Tip: Under View->Editor you can make geany show whitespace, which makes these errors easier to see. You might also want to visit the Document window and have geany wrap lines for readability.

Lists in Python can contain data of multiple types. Try adding `300` and `.45` without any quotes to the list. Change the print statement to:

```
      print('The title is {title} and the type is {tp}'.format(title = m, tp = 
type(m).__name__))
```

In our output, we see that the last two entries in the list are of the `int` and `float` types respectively.

Let's change the way we iterate through the list. Replace the for loop by:

```
      for i in range(len(apps)):
            print(str(i) + ": " + str(apps[i]))
```

The list `apps` is accessible through the square brackets and is 0 based (index of list starts at 0). Within the `print` statement we still have to cast all of the entries to a string using `str()`, or else Python gets confused about concatenating strings with integers or floats. The `len()` function returns the length of a list, the `range` function will return an iterable list from 0 up to N – 1, if N is the argument.

Remove the .45 entry by typing `apps.pop()` below where apps is declared. Then remove 300 by typing `del apps[5]` . The `pop` command returns the final element from the list (and removes it), while `del` just removes the indicated index as you'd expect. Add a new entry by typing:

```
      apps.append('A Liar\'s Autobiography')
```

Now that we have some of the basics down we can go on to actually building a pairing algorithm.

1.1.5) Making a distance function (i.e., a type of similarity measure)
Our purpose here is to get an understanding of recommender systems. A

recommender system is a system that automatically predicts user preferences, usually in the form of items that a user might like or be interested in.

Collaborative Filtering (CF) is a class of methods that recommend items to users based on the preferences other users have expressed for those items. In other words, a CF recommender system exploits past behavior of the user. For example, bol.com will recommend books to you based on which books you have purchased in the past. In this lab "past behavior" is taken to be the ratings that people have previously assigned to apps. These ratings are used to predict which apps people might like in the future.

User-user collaborative filtering, a widely used algorithm studied in this course, predicts a movie for a user by first looking for similar users. It uses rating patterns to find similar users. Note: The "People who watched X also watched Y" algorithm is called item-item collaborative filtering, and is also widely used.

In this lab, we will implement a reciprocal recommendation algorithm that does not recommend apps or other items to users, but rather recommends users to each other. We will use these recommendations to determine which pairs of people will be partners for the lab in this course. An important part of this course is for you to learn how algorithms relate solutions to real life problems that affect real people. During this assignment, you should think of an idea for a pairing not only from the algorithm perspective, but also from a user perspective. In other words, which aspects of the algorithm might be most important to users?

First, we implement a very simple way of calculating pairwise similarity between users, where a user's preferences are represented as a vector. Instead of movie rankings, we will be using flavor rankings. We then make the assumption that flavor preferences reflect characteristics of people that are related to their potential compatibility as lab partners. On the basis of this assumption we divide our set of users into pairs.

For both steps of this week's lab you work individually; however, discussion is allowed. At any point in this lab you should ask the instructors (this is the set of people to whom you also gave access to your lab report above) if you have questions. You can also ask other students.

First, we will be learning about functions.

Python allows for the importing of many modules that contain lots of functionality so you don't have to keep reinventing the wheel. Right now however, we will not be using most of that functionality so we can learn some more about Python.

From the previous exercises, remove all of the code except the line that prints the exercise title. Next define a Euclidean distance function as follows:

```
def euclidean_distance(p,q):
        # this is a comment
        # your code goes here
```

Like with the loops, the indentation again specifies the scope of the function. You don't need to specify the return type or parameter types of the function like you would in Java. Note that the same worked for the `apps` variable earlier. For this distance function, we're going to assume a few things. Both parameters `p` and `q` are n dimensional vectors, of the same dimension which is greater than 0. The equation for Euclidean distance in n dimensions is:

$$d(p,q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \ldots + (p_n - q_n)^2}$$

There are many ways to implement this, right now we'll go for the straightforward way:

```
def euclidean_distance(p,q):
    d = 0
    for i in range(len(p)):
        d += (p[i] - q[i])**2

    return math.sqrt(d)
```

For this to work, we will have to import the Python math module which can be done by adding `import math` to the top of the file. This will import the whole math module, even though we'll only need the square root function. To only import the square root function, we could use `from math import sqrt`. In that case, you could use the square root function by typing `s = sqrt(25)` or even `from math import sqrt as square_root` (which we would call as `s = square_root(25)`) if you want to change the name of the function implementation.

We can test the new function by printing its outcome, for example by using:

```
a = [4,8,1,5,1,6,2,3,4,2]
b = [3,2,2,3,9,4,0,4,2,6]
print euclidean_distance(a,b)
```

Although you might want to use some smaller or simpler vectors to check for correctness.

We want to control the effects of the relative vector magnitudes on their distance, therefore we introduce a normalization step.

Assume for now that all values in the vectors range from 0 to 10. In that case, we only have to divide each entry by that maximum. Note that there are many ways to normalize values, the ‘correct’ way depends on the application. In lecture, for example, we discussed normalizing vectors with their lengths to obtain unit vectors. Here, for variety, we normalize with the value of the maximum component.

```
    def normalize_vector(p, maximum):
        return [float(x) / maximum for x in p]
```

Note the way in which we iterate through `p`. This illustrates a powerful feature of Python called list comprehension.

In the block below, you'll find a list of vectors that you can easily copy-paste into the code. We'll put the feature vectors into a list itself. By the way, a list can hold lists, even of unequal lengths, although that is not applicable to our situation right now.

```
    [10.0, 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0]
    [9.0, 8.0, 9.0, 8.0, 9.0, 7.0, 4.0, 2.0, 9.0, 5.0]
    [9.0, 7.0, 6.0, 7.0, 4.0, 4.0, 7.0, 1.0, 7.0, 4.0]
    [3.0, 6.0, 5.0, 3.0, 4.0, 3.0, 5.0, 7.0, 8.0, 8.0]
    [4.0, 5.0, 4.0, 2.0, 5.0, 6.0, 4.0, 6.0, 7.0, 8.0]
    [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]
```

We can only use `normalize_vector` *below* where it has been defined, so at the bottom of the code add:

```
    M = []
    for m in [a,b,c,d,e,f]:
        M.append(normalize_vector(m, 10.0))
```

`M` is the list in which we gather all the individual vectors. `M = []` means that M is initialised as an empty list. The letters are the elements, which are other lists in this case.

1.1.6) Calculating the distance matrix

There are six vectors, so if we're not so smart we'll be building a 6 by 6 distance matrix ending up with 36 entries. However, since Euclidean distance is symmetric and we don't care about a vector's distance to itself, we actually only need to fill out part of the distance matrix. The `range` function is useful in this situation:

```
    def distance_matrix(vectors):
        result = []
        N = len(vectors)
        for i in range(N):
            for j in range(i+1,N):
                ind = (i,j)
                dis = euclidean_distance(vectors[i], vectors[j])
                result.append( (ind, dis) )

        return result
```

A few things are useful to remember here. The use of tuples, because it is faster and safer (as they are immutable). The `range` function can run from a lower bound to the non-inclusive upper bound. Python control flow functionality can be seen here. Note that control flow statements end with a colon "`:`" and indentation is used to demarcate their scope. To see control flow at work, add the following to the bottom of the file:

```
D = distance_matrix(M)
for (i, d) in D:
      print(i,d)
```

`result` was constructed to be a list of 2-tuples, which were specified as `(ind, dis)`. In this way we can specify any n-tuple we'd like. We can then simply tell Python to iterate over the list of tuples we expect to find in `D`. There are only 15 distances specified, rather than the 36 we'd find in a matrix. If we remember our combinatorics this is unsurprisingly equal to 6 choose 2 or $\binom{6}{2}$

The output should resemble (the actual distances will differ):

```
Exercise 1 - Pairer
(0, 1) 0.6382847385042254
(0, 2) 1.4948471163415233
(0, 3) 1.625415426480866
(0, 4) 1.0243938285880987
(0, 5) 1.1547005383792515
(1, 2) 1.227262335243029
(1, 3) 1.3471506281091268
(1, 4) 1.030402055055078
(1, 5) 1.4782371884055634
(2, 3) 1.4782371884055636
(2, 4) 1.8324913891634047
(2, 5) 1.972026594366539
(3, 4) 1.9404721329525534
(3, 5) 2.084629226588191
(4, 5) 1.4185717038670782
```

Run `print(D)` to look at the less readable form, to get an idea of how the list of tuples works.

1.1.7) Matching the pairs

We have a list of 15 unique (pairing, distance) tuples. The next step would be to identify which of those pairs represent similar users. Shorter distance implies greater similarity, therefore it would be useful to sort our tuples according to their distance element. Sorting a list can be done using `sort()` which modifies the list and returns `None`; though right now we have a list of tuples, so we need to make clear on which value to sort. For this reason, we'll use the function `sorted()` instead. Overwrite the for loop below the declaration of D with the following:

```
D_sorted = sorted(D, key=lambda D: D[1])
for (i,d) in D_sorted:
      print(i,d)
```

The built-in `sorted` function takes a key function, which it uses to sort. The value of the key function should be a function that returns a single key that can be used for sorting. Here we use a lambda function on list `D`, which extracts the value of the second (index 1) entry of the tuple.

The output should look like:

```
Exercise 1 - Pairer
(0, 1) 0.6382847385042254
(0, 4) 1.0243938285880987
(1, 4) 1.030402055055078
(0, 5) 1.1547005383792515
(1, 2) 1.227262335243029
(1, 3) 1.3471506281091268
(4, 5) 1.4185717038670782
(1, 5) 1.4782371884055634
(2, 3) 1.4782371884055636
(0, 2) 1.4948471163415233
(0, 3) 1.625415426480866
(2, 4) 1.8324913891634047
(3, 4) 1.9404721329525534
(2, 5) 1.972026594366539
(3, 5) 2.084629226588191
```

Looking ahead to what is to come: When you make an actual matching algorithm, you might want to consider what measure you want to optimize. For example, do you want to make it so that there is *no* pairing for which every participant is equally satisfied or better off (Pareto optimal), or do you want to optimize such that the worst pairing is still as good as possible (Egalitarian). Maybe you want neither of those but include some other neat feature?

Now back to where we are now: Right now, we are going for a greedy approach.

Put the output of the distances into your report. In your report, also explain briefly which pairs you would expect would be recommended by a greedy algorithm. (Do this without looking ahead.)

Moving on: We allocate a pair from our sorted list based on closeness. The participants from that pairing are then no longer eligible to be paired with another participant. We need to do a bit of bookkeeping, so we don't match a participant that is already in a pair:

```
free_participants = [0,1,2,3,4,5]
pairs = []
```

```
	D_sorted.reverse()

	while len(free_participants) > 0:
		# add next closest pair
		closest_pair = D_sorted.pop()[0]
		pairs.append(closest_pair)

		# indicate participants are now paired
		p = closest_pair[0]
		q = closest_pair[1]
		free_participants.remove(q)
		free_participants.remove(p)

		# remove the newly paired participants from the distance matrix
		for i in range(len(D_sorted)-1,-1,-1):
			if p in D_sorted[i][0] or q in D_sorted[i][0]:
				del D_sorted[i]

	for p in pairs:
		print p
```

A few things happen here. `D_sorted` is reversed, because if we're using `pop()`, we want to get the smallest distance first. For the pairings, we only want the pair tuple, not the distance (though there would be nothing wrong with including that). We can simply put a `[0]` behind the `pop` command and Python will understand that we mean the first entry of the tuple that was just popped. The `range` function now iterates from the final element of `D_sorted`, down to 0. That is, the first parameter indicates the starting number, the second indicates the step, and the last indicates the non-inclusive boundary (hence `-1` instead of `0`). Within the conditional, we use `in` to check if `p` or `q` are contained in the entry we're looping through in reverse. We might have been tempted to use an iterator construction like: `for tup in D_sorted:` , however, you can't remove entries from a list while you're iterating through it.

At this point we have a very basic pairer based on an arbitrary toy example of six participants. Clearly coding like this normally is a bad idea. Next up, we'll be extending it to look more like a proper program, which can use participant IDs, does some exception handling, can load the participants from a file, and that uses modules.

1.1.8) Loading from a file

The students flavor ratings and a unique identifier can be found here
https://drive.google.com/file/d/16zQbSly4JLgejCQr57E1DiKhaaT2Zikp/view?usp=sharing

Each line of the file represents one student, and the line format is:

```
id,app1rating,app2rating,...,app50rating
```

whereby `appxrating = {DK|1|2|3|4|5}` or is empty.

Copy it to the `Exercise 1` folder (try it with the `cp` command in the terminal) and open it in LibreOffice Calc and see if you can understand how it works. You might want to change the column width (under Format->Column->Width) to a few centimeters. By looking at the layout of the file, we see that the first row is a header containing titles and descriptions. The user identifier is found in the tenth column, while the columns following that contain the flavor ratings.

Comma separated files are very common, so Python has a way to read them. Make a new file '`matcher.py`' and include the following code:

```
import csv
with open('Sheet_1.csv', 'rb') as csvfile:
      reader = csv.reader(csvfile, delimiter = ",", quotechar="\"")
      header = reader.next()
      identifiers = []
      scores = []
      for row in reader:
            # Store the identifier and corresponding scores
            identifiers.append(row[0])
            scores.append(map(int, row[1:]))
```

Using `with` and `open` in this way, file opening and especially closing is done automatically. These things can also be done manually if you like. Calling `reader.next()` returns a row each time, and we can also easily iterate through those rows with a for loop construction.

Try running this code and observe what happens. You might get an error specifying that an empty string cannot be mapped to an integer. Look again at the csv file and check why there might be an empty string.

The `map` function applies a function, in this case `int()` to each item in an iterable. For the ratings, we specify that it should be kept the same if the string is not an empty string, or else it should be set to the string `'0'`.

There is still another change we could make, if we assume that the named identifiers are indeed unique. Python has good support for dictionaries (and they are much easier to be used than in Java), and that data structure is suited for our situation now.

```
scores = {}

for row in reader:
      # Store the identifier and corresponding scores
      identifier  = row[0]
      ratings     = map(int, [x if x is not '' else '0' for x in row[1:]])
      scores[identifier] = ratings
```

```
for key, value in scores.iteritems():
      print key, value
```

By typing `scores = {}`, we indicate that scores becomes initialized to an empty dictionary. The dictionary can be filled as shown in the loop. *In case you should be using Python 3 in the future, please note that this way of using print is specific to Python 2.* The `iteritems()` returns an iterator over the key, value pairs in scores. A dictionary works as you'd expect. Items can be retrieved or overwritten using `dictionary_variable_name['Key_Name']`.

After you've looked at the contents of the output terminal you can remove the printing lines. Next we'll move on to implementing cosine similarity.

1.1.9) Cosine Similarity

Consider the formula for cosine similarity between two vectors. n is the number of components in the vectors, which must be the same:

$$sim(x,y) = \frac{x \bullet y}{\|x\|\|y\|} = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} {x_i}^2} \sqrt{\sum_{i=1}^{n} {y_i}^2}}$$

We see that the numerator is the dot product, i.e., a sum over the element-wise multiplication between two vectors. Trivially, we could implement this using a for loop, such as:

```
def cosine_similarity(p, q):
      denominator = 0
      for i in range(len(p)):
            denominator += p[i] * q[i]
      # The rest of the code
```

However, we already dealt with loops so now let’s use NumPy, a Python math library that is written in C for speed, is very suited for this kind of mathematical code. (From wiki:) NumPy is an extension to the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large library of high-level mathematical functions to operate on these arrays.

Below `import csv` add `import numpy`. Normally if we're using modules we might need to install them first. That step has already been taken care of in the VM. If you are not already, you will soon be convinced that NumPy is well suited for vectorized mathematical computations.

Using `numpy`, we can implement cosine similarity as follows:

```
import numpy as np
def cosine_similarity(p, q):
      numer = np.sum(np.multiply(p,q))
      denom = np.sqrt(np.sum(np.square(p)))*np.sqrt(np.sum(np.square(q)))

      if denom is not 0:
            return float(numer) / denom
      else:
            return 0.0
```

Note that we use the convention where `numpy` is called as `np` to save some space on the line.

Foreshadowing topics to come, we provide the following example: using a `numpy` array we can work on array elements in the same way as we might in Matlab. Suppose we have an image in RGB format, that has the dimensions N by M by 3, and we want to zero the R channel.

```
# im contains the NxMx3 matrix
im_numpy = np.array(im)
im[:,:,0] = 0
```

Back to the problem at hand: since the cosine distance is common distance measure, we could have also simply used:

```
from scipy.spatial.distance import cosine
```

to compute the cosine similarity. `scipy` is a module which is commonly used in scientific computations. We can now use this to check the correctness of our implementation:

```
A = scores['piepe']
B = scores['jelen']
print "own: \t", cosine_similarity(A, B)
print "check: \t", 1 - cosine(A, B)
```

We use the `1 - cosine(A,B)` since we would like a *similarity*, and the `scipy` implementation expresses *distance*, cf.
http://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.spatial.distance.cosine.html#scipy.spatial.distance.cosine

1.1.10) Exploring the differences between similarity functions

In this exercise, we move on to writing more Python code, and at the same time we learn about similarity functions. In particular, we explore:

- Impact of normalization on the similarity function.

- Impact of mean-centering on the similarity function.
- Impact of item overlap on Pearson correlation.

The adjusted cosine similarity is the similarity over "mean-centered vectors" which means that you adjust each vector by subtracting the average of all its components.

Write a Python script that outputs one line for each of the 15 pairs of vectors given in the assignment. The form should be:

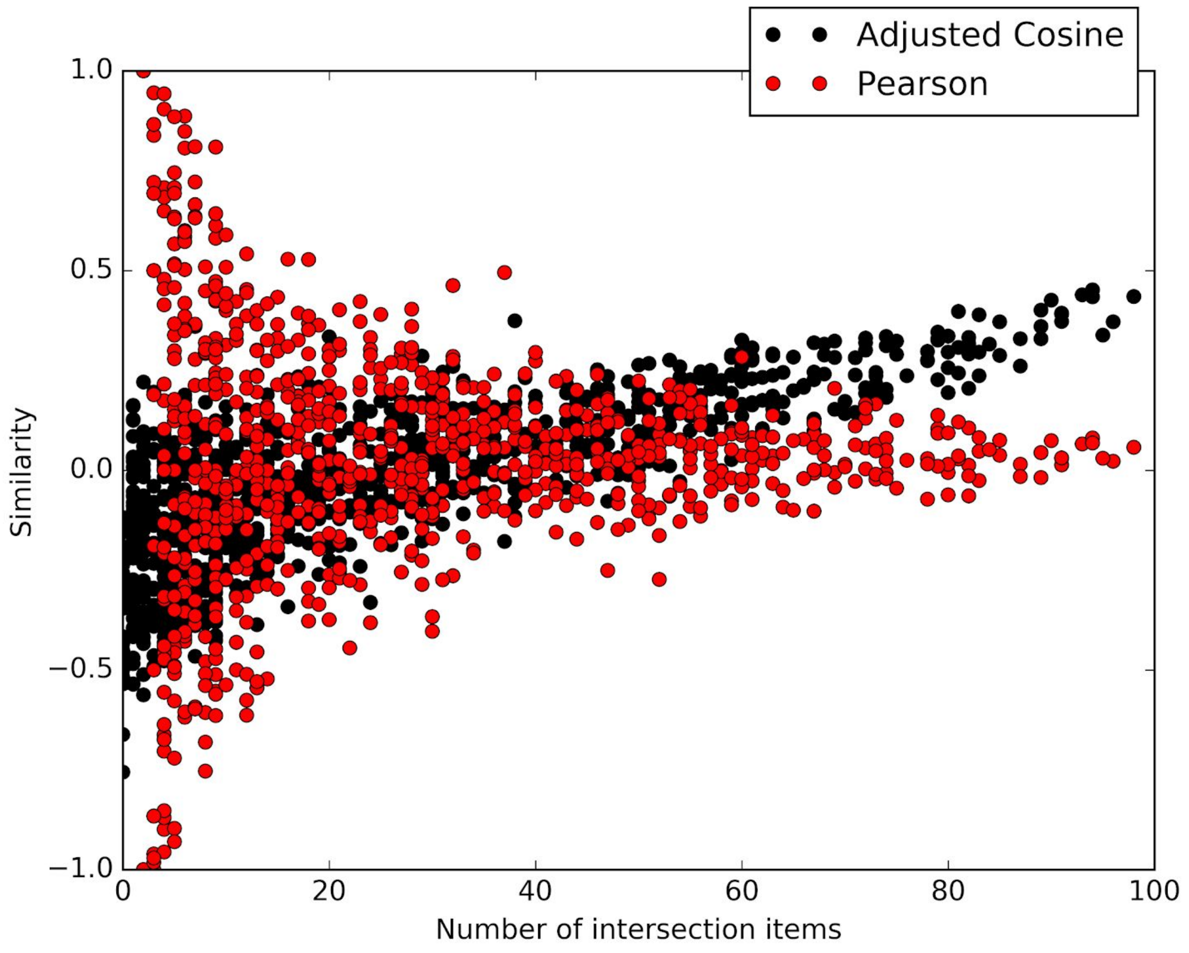
v1_index v2_index dot_product cosine adjusted_cosine

Insert this output into your lab report.

Now, look at the vectors for which the dot_product or the adjusted_cosine give you significantly different patterns than the cosine.

In your lab report, provide an informal description of the difference between the two vectors in a case in which the dot product and the cosine differ dramatically. Do the same for the cosine and the adjusted cosine.

Next, the graph below shows the similarity measure between pairs of users for the Adjusted Cosine and for Pearson as the number of co-rated items (the number of items that they have both rated) increases. Please look at the graph and answer the questions below.



Which similarity measure is more sensitive to the number of items in the intersection?

1.1.11) Create a pairing algorithm

Finally, we reach the “Pairing Assignment” in which you create a pairing algorithm using Python. It should operate on the same format as the csv file we've been working with. It should be able to return the ranked matches for a given csv file with flavor rankings. Use method of your own choice to create the list of pairs. You should strive to improve on greedy matching. Also, your pairing algorithm should be able to elegantly handle an odd number of students by creating one group of three.

The input will be of the same form as Sheet_1.csv. You may assume that the identifiers are unique.

Its output should be a plain text file with human readable pairings. Scores do not have to be normalized, but if you would like to normalize them, that’s fine. The best match should be on top, the worst at the bottom. Example:

```
1. oonger 1.00 (Boon, Jager)
2. sonung 0.91 (Larson, Hung)
3. eraiou 0.91 (Cabrera,Demetriou)
4. linese 0.40  (Palin, Cleese)
5. mandle -0.45 (Chapman, Idle)
```

Pairing Assignment implementation: Give a bit of thought to what would constitute an interesting and acceptable pairing for your classmates. You might need some time out of lab for the implementation, but you should make a good start now.

Pairing Assignment pitch: Create a pitch for your algorithm of 100 words. This pitch should explain your algorithm briefly and clearly, and also "sell" the best features of your algorithm to your fellow students. Remember to mention in your description the way in which your pairing algorithm seeks to deliver the best possible pairings for the users of the pairing recommender system. In other words, why should the users be particularly happy with these pairs?

You are welcome to use the following similarity and distance metrics in your pairings. These are the ones that were covered in class.

Similarity and distance measures
See also the slides from the first lecture in Week 1.

Let n be the number of dimensions in the vector.

1. Boolean similarity: The count of the symbols types common to two strings.
2. Jaccard similarity coefficient
$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}.$$
3. Taxicab distance
$$d_1(\mathbf{p},\mathbf{q}) = \|\mathbf{p}-\mathbf{q}\|_1 = \sum_{i=1}^{n} |p_i - q_i|,$$
4. Euclidean distance
$$\mathrm{d}(\mathbf{p},\mathbf{q}) = \mathrm{d}(\mathbf{q},\mathbf{p}) = \sqrt{(q_1-p_1)^2 + (q_2-p_2)^2 + \cdots + (q_n-p_n)^2}$$
$$= \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}.$$
5. Normalized Euclidean distance (n as the number of points)

d(p,q) = $\sqrt{\sum_{i=1}^{n}\left(\frac{q_i}{\|q\|}-\frac{p_i}{\|p\|}\right)^2}$

6. Dot Product

$$x \bullet y = \sum_{i=1}^{n} x_i y_i$$

7. Cosine similarity

$$sim(x,y) = \frac{x \bullet y}{\|x\|\|y\|} = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^{\,2}}\sqrt{\sum_{i=1}^{n} y_i^{\,2}}}$$

8. Adjusted cosine similarity (i.e., cosine similarity on mean-centered vectors) Here, it is expressed for the case of a user item matrix containing ratings. Recall from lecture that I_u and I_v are the sets of items (in the case of the pairing algorithm, this is apps) that users u and v, respectively, have rated.

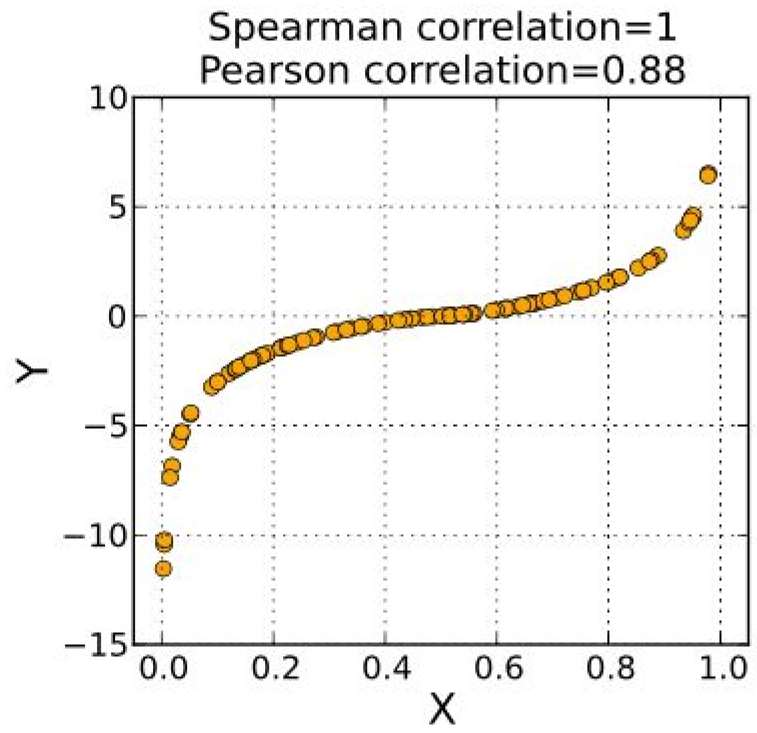
$$sim(u,v) = \frac{\sum_{i=1}^{n}(r_{u,i}-\bar{r}_u)(r_{v,i}-\bar{r}_v)}{\sqrt{\sum_{i=1}^{n}(r_{u,i}-\bar{r}_u)^2}\sqrt{\sum_{i=1}^{n}(r_{v,i}-\bar{r}_v)^2}}$$

9. Pearson correlation

$$sim(u,v) = \frac{\sum_{i\in I_u \cap I_v}(r_{u,i}-\bar{r}_u)(r_{v,i}-\bar{r}_v)}{\sqrt{\sum_{i\in I_u \cap I_v}(r_{u,i}-\bar{r}_u)^2}\sqrt{\sum_{i\in I_u \cap I_v}(r_{v,i}-\bar{r}_v)^2}}$$

10. Spearman rank correlation

$\rho = 1 - \frac{6\sum d_i^2}{n(n^2-1)}.$ where $d_i = x_i - y_i$ is the difference between rankings of item i by participants x and y.



Comparison of Spearman correlation and Pearson correlation for two variables X and Y (i.e., a set of pairs of observations (x,y))
https://en.wikipedia.org/wiki/Spearman%27s_rank_correlation_coefficient

There is a specific procedure for submitting the Pairing Assignment. Please make sure that you carry out both of these steps:

1. As for the other assignments, put **the code** and **the pitch** for your pairing algorithm into your report
2. Please submit your code as a text file on Brightspace under "Pairing algorithm" Add your name and also your pitch as a comment at the beginning of the code.

The Pairing Assignment must be submitted by 23.59 on Tuesday 19 Feb.