Python for Collective Intelligence and Collaborative Filtering

Jeremy Patrick Johnson  
Co-Founder Python Malaysian Users Group  
jeremy@python.my

Abstract  
This paper will define the two terms Collective Intelligence and Collaborative Filtering and discuss how these two ideas can be used to create personally relevant filters allowing end users more personalized access to information on their chosen topics of interest. In addition various mathematical models used to filter data and compare preferences and their corresponding pythonic implementations will be discussed. Finally a simple example using web API’s and Collective Intelligence algorithms will be demonstrated to provide an idea of the type of things that can be achieved, relatively easily, using python for Collective Intelligence and Collaborative Filtering. This short abstract will be accompanied by a talk given at PyCon Asia 2010.

1. Introduction  
These days there is so much information online that finding exactly what you are looking for can be a time-consuming challenge. While Google is an excellent service and it goes a long way in assisting end-users with filtering through the myriad of data on the Internet, its organizational strategy focuses on the tastes or preferences of the masses. This paper will present organizational strategy focused on personal preference. But first we must define some terms:

Collective Intelligence as defined in Wikipedia is – “a shared or group intelligence that emerges from the collaboration and competition of many individuals”

Collaborative Filtering (also defined by Wikipedia) is – “the process of filtering for information or patterns using techniques involving collaboration among multiple agents, viewpoints, data sources, etc.”

These two paradigms can be brought together to help create personalized filters for end-users to assist with the never-ending quest to find meaningful and individually relevant information on the Internet.

2. Combining Collective Intelligence and Collaborative Filtering  
If Collective Intelligence is the idea that the whole is smarter than the individual, Collaborative Filtering is the idea, that the knowledge of the whole can more accurately and
efficiently organize and filter data. While these approaches are effective they tend to produce organizations that generally reflect the societal norms. For example you find more programming links about Java or .NET than Python on del.icio.us because of their relative popularity. The aim here is to remove the lens of popularity and replace it with the lens of personal preferences. In such a way a Python developer might find more links to Python sites, while a LISP developer would find more links on LISP. The whole concept can be thought of as a personalized search.

2.1 Common Examples
Some common examples of this concept include; asking your friends for restaurant recommendations, or asking a group of python developers for IDE recommendations as opposed to a group of programmers from all languages. More mainstream examples might be reddit.com or del.icio.us, last.fm, amazon.com’s recommendation system, and even Google’s new personalized search product. So the examples are out there, now how do we do it?

2.2 Prerequisites
The process of Collaborative Filtering requires a few pre-requisites. First you must have a group of people or items that will provide ratings to a set of things. The set of things can be anything from cars, to restaurants, to music, to political parties. As an example let’s use the del.icio.us API to get a list of users who frequently create bookmarks on a tag (say python):

```python
from pydelicious import get_popular, get_userposts, get_urlposts

def initializeUserDict(tag, count=5):
    user_dict = {}
    # Get the top count popular posts
    for p1 in get_popular(tag=tag)[0:count]:
        # Find all users who posted this
        for p2 in get_urlposts(p1['url']):
            user = p2['user']
            user_dict[user] = {}
    return user_dict
```

So this will return user_dict, which will contain a list of users who posted recently on the particular tag.

Preferences are assigned to the set of things in a standard way, for example a rating of 1 to 10 or a yes no vote. The exact system of ratings / preferences is not important as long as it can be converted to a normalized numerical value. So for our delicious example we’ll get a list of the recent urls posted. Then for each user they get a score of one if they also posted that link or zero if they didn’t post the link.

```python
def fillItems(user_dict):
    all_items = {}
    # Find links posted by all users
```
for user in user_dict:
    posts = get_userposts(user)
    for post in posts:
        url = post['url']
        user_dict[user][url] = 1.0
        all_items[url] = 1

# Fill in missing items with 0
for ratings in user_dict.values():
    for item in all_items:
        if item not in ratings:
            ratings[item] = 0.0

(Note: error handling omitted for brevity) Here we get a list of posts for each user. The user
who posted the URL gets a score of 1 for that url, meaning they posted it. The second loop
goes through all the urls and checks to see if a user posted that url, if they didn’t that user is
assigned a score of 0 for the particular post. So at the end each user will have a list of all
urls with either a 1 (they posted it) or a 0 (they didn’t post it) associated.

So given the set of people and the set of their preferences it is then the task of Collaborative
Filtering to determine which people have similar preferences and use that “Similarity Score”
(Segaran, 2007) to influence recommendations given out by the system.

The crux of Collaborative Filtering lies in effectively filtering data, and matching
preferences. The tricky part is the “effective” point, and like all good problems there is no
one solution that fits all. Therefore a couple of techniques are presented below to give the
reader an idea of potential algorithms that may be useful. (Source code for these techniques
will be presented in the accompanying talk.)

3. Filtering Techniques
There is a multitude of filtering techniques from the straightforward to the outright bizarre
that can be used. Providing a broad survey of the various techniques is beyond the scope of
this paper. However, to serve as an introduction to the field of Collaborative Filtering two
algorithms that are both fairly simple to understand are described below. (Others can be
found here: http://en.wikipedia.org/wiki/Metric_%28mathematics%29#Examples)

3.1 Euclidean Distance
“In mathematics the Euclidean distance or Euclidean metric is the "ordinary" distance
between two points that one would measure with a ruler, and is given by the Pythagorean
Formula.” (Wikipedia, 2010).

In order to use the Euclidean Distance we chart personal preferences, using the “set of
things” that have been ranked as the axis to the chart. Example below:
In this case the X-axis and the Y-axis would each represent something in the list of things being ranked. For example if you are ranking programming languages the X-axis might represent “Python” and the Y-axis might represent “Java”. Or perhaps “Dell” and “Acer” if you were ranking PCs.

Applying the Pythagorean formula to this data set will give us the distance between two users preferences. We want a rating of 1 to mean that the users have the exact same preference and 0 to mean complete opposite preferences. That is however the opposite result of what is produced by the Pythagorean formula so we invert it (but add 1 first to avoid a divide by zero). The code looks like this:

```python
from math import sqrt

def euclidean_distance(prefs,p1,p2):
    si={}
    for item in prefs[p1]:
        if item in prefs[p2]:
            si[item]=1

    #if they have no ratings in common, return 0
    if len(si)==0: return 0

    # Add up the squares of all the differences
    sum_of_squares=sum([pow(prefs[p1][item]- prefs[p2][item],2) for item in prefs[person1] if item in prefs[person2]])

    return 1/(1+sum_of_squares)
```

In the above code example prefs would be the list of items being ranked. So from our running delicious example prefs would be the filled user_dict, or the list of urls that each user posted. We pair users and loop through what they posted, if all urls posted were identical the euclidean_distance function would return 1.
3.2 Pearson Correlation Coefficient

While the Euclidean distance score is easy to understand it doesn’t always create the best similarity score. For the case where users have similar preferences but user A consistently gives higher ratings than user B the Euclidian distance score will not produce accurate results. However the Pearson Correlation Coefficient will factor out that bias because it focuses on measuring the data’s fit to a straight line. It will return a number between 1 and -1 with a value of 1 indicating two users have rated all items the same. It’s equation is below (Wikipedia,2010):

\[
P_{a,u} = \frac{\sum_{i=1}^{m} (r_{a,i} - \bar{r}_a) \times (r_{u,i} - \bar{r}_u)}{\sqrt{\sum_{i=1}^{m} (r_{a,i} - \bar{r}_a)^2 \times \sum_{i=1}^{m} (r_{u,i} - \bar{r}_u)^2}}
\]

Figure 3.2. Pearson Correlation Coefficient Formula.

So for cases where data is less normalized then the Pearson Correlation Coefficient is a good technique to use to produce a “Similarity Score”. The code is shown below:

def pearson_correlation(prefsMapping, p1, p2):
    '''returns the pearson Correlation Coefficient, which is a measure of how similar p1 and p2's preferences are to one another. Value is between 1 and 0 with 1 meaning p1 and p2 have identical preferences'''
    #Get the list of items rated by both parties
    itemRatings={}
    for item in prefsMapping[p1]:
        if item in prefsMapping[p2]: itemRatings[item]=1 #both parties rated the items

    #find the number of elements
    n=len(itemRatings)

    # if there are no ratings in common, return 0
    if n==0: return 0

    # Add up all the preferences
    sum1=sum([prefsMapping[p1][it] for it in itemRatings])
    sum2=sum([prefsMapping[p2][it] for it in itemRatings])

    #sum up the squares
    sum1Sq=sum([pow(prefsMapping[p1][it],2) for it in itemRatings])
    sum2Sq=sum([pow(prefsMapping[p2][it],2) for it in itemRatings])

    #sum up the products
    pSum=sum([prefsMapping[p1][it]*prefsMapping[p2][it] for it in itemRatings])

    return pSum/(sqrt(sum1Sq*sum2Sq))
# Calculate Pearson score

```python
num = pSum - (sum1*sum2/n)
den = sqrt((sum1Sq-pow(sum1,2)/n)*(sum2Sq-pow(sum2,2)/n))
if den==0: return 0
r = num/den
return r
```

For our del.icio.us example the extra work involved with the Pearson Similarity Score is not necessary. Because our ratings are either 0 or 1 the data does not need to be normalized. However if your dataset contained different rating systems, say perhaps half of the movie critics used the 5 star rating system and the other half used a 1-10 rating system, then the Pearson Similarity Score would most likely provide a better result. There are other scoring techniques available beyond the scope of this short abstract. Some other scoring systems to consider might be the Tanimoto Coefficient or Manhattan Distance.

## 4 Making Recommendations

Once you have figured out a “Similarity Score” then you want to make recommendations based upon that score. The easiest solution is to choose the person with the highest similarity score and just provide his recommendations. However a more effective way would be to use the “Similarity Score” as a weight and apply it to each of the ratings. For example imagine if you had a data set of user recommendations for athletic apparel. If John provides the following raw ratings (Nike = 8, Puma=9, Addidas=2) and he had a similarity score of .5 then his weighted ratings would be (Nike=4,Puma=4.5, Addidas=1). So then we can choose recommendations across the entire set based upon the new weighted rankings and just provide those recommendations with the highest rankings, irrespective of who made them. This is what was earlier referred to as “to remove the lens of popularity and replace it with the lens of personal preferences”.

So coming back to our running del.icio.us example. We have thus far, built a dataset of url that have been posted to del.icio.us and the people who posted them. We have determined that the Euclidian Distance would be a simple way to create a similarity score between two people in our dataset. So now lets build a function that loops through the entire dataset and calculates the Euclidian Distance for all members. In effect we will be comparing one person (the person getting the recommendations) to everybody else in the dataset. The following function does exactly that.

```python
def findPeopleWithSimilarInterests(prefs, subject, 
    scoringAlgorithm=euclidean_distance, numResults=5):
    '''call the scoring Algorithm for everybody in the prefs
dataset and compare that person to subject. The return the
dataset ordered by similarity to subject'''
```

- 6 -
scores=[(scoringAlgorithm(prefs,subject,otherPerson),otherPerson)
        for otherPerson in prefs if otherPerson != subject]

#sort by similarity with most similar on top
scores.sort()
scores.reverse()
return scores[0:numResults]

Here the passed in parameters are:
- prefs - the dataset of users and items and their ratings
- subject - the person being compared against
- scoringAlgorithm - the algorithm to generate the similarity score, i.e. Euclidean_distance, or pearson_correlation
- numResults - the number of results to return

So the idea is you pass in the dataset and the name of the user you want to rate and the function will return a dictionary of the name of the user and the users similarity score ordered by the strongest similarity. For del.icio.us this could serve to find other people who have similar interests to the current user (assuming tagging a link in del.icio.us denotes interest). Or it could be taken a step further by applying the similarity score for the user as a weight and then using that weight to recommend links, which would provide a list of links to the users that should be similar to the users preferences.

So now that we have all the pieces needed to build a working example of a del.icio.us link recommender let’s put them together in the following example.

4 Real World Example
As a recap we are going to do the following.

1. Build a data set of users and posted urls using the del.icio.us api.
2. Match preferences between a user and all other users (i.e. create a similarity score) using one of the two formulas in section 3 above.
3. Using the similarity score we can weight the tagged links and present those links with the highest weighting to our blog reader. The more accurate our filtering technique is the more personalized the recommendations will be.

So lets get started:

    >> pythonistas = initializeUserDict(‘python’)
    >> pythonistas[jiz0]={} #Add myself 😊
    >> fillitems(pythonistas)
Ok at this point I have my initial dataset, which consists of a list of users that have recently posted links with the tag python (plus myself). Further each user has a list of each url that they posted. So now I want to find people with similar interests to myself.

```
>> findPeopleWithSimilarInterests(pythonistas, 'j1z0')
```

This will compare the url posts in my account (j1z0) with those of all the other users in the pythonistas dataset. The results (keep in mind this is live data so your results will be different):

```
[(0.058823529411764705, u'broccolini'),
 (0.052631578947368418, u'zee8'),
 (0.052631578947368418, u'yukiex'),
 (0.052631578947368418, u'yaanno'),
 (0.052631578947368418, u'y Ug')]
```

So as we can see the top 5 people most like me are listed with their similarity scores. It seems that I’m quite unique cause nobody is that much like me. ☺ Never the less I could simply look up these users del.icio.us page and pick links or we can take this example one step further and use the similarity score for each user as a weighting that is applied to each link and then recommend links with the highest weighted score. Lets do that here:

```python
def getRecommendations(prefs, subject, scoringAlgorithm=euclidean_distance):
    totals={} weights={}
    for otherPerson in prefs:
        if otherPerson==subject: continue
        score=scoringAlgorithm(prefs,subject,otherPerson) #ignore zero scores
        if score<=0: continue
        for item in prefs[otherPerson]:
            #only score items I haven't rated (i.e. not viewed)
            if item not in prefs[subject] or prefs[subject][item] == 0:
                #store the weighted score for the item in totals
                totals.setdefault(item,0)
                totals[item]+=prefs[otherPerson][item]*score
                weights.setdefault(item,0)
                weights[item]+=score

    #take the average weights so popular sites are rated higher
    rankings=[(total/weights[item],item) for item,total in totals.items()]
    rankings.sort()
    rankings.reverse()
    return rankings
```
In the above code we through the dataset and determine the similarity score between subject and the otherPerson. Then we use that similarity score as a weight multiplying it by the rating for the item (in this case ratings are always 1 or 0 but ratings of 0 are ignored so in effect the weight becomes the rating score) as seen in this line:

\[
\text{totals[item]} += \text{prefs[otherPerson][item]} \times \text{score}
\]

After getting the weighted rankings of all the users, we average out each individual ranking so that sites that popular urls are always what is recommended (Remember we don’t want Google results we want personal results.) After the average is taken the results are sorted and return.

With that function in place we can then ask for a list of recommended links for j1z0:

```python
>> getRecommendations(pythonistas, ‘j1z0’)[:10]
```

Limiting the results to only ten, we get the following output:

```
[(0.18189233278955932, u'http://learnpythonthehardway.com/index'),
 (0.16639477977161485, u'http://learnpythonthehardway.org/index'),
 (0.16639477977161485, u'http://fitzgen.github.com/zoolander/'),
 (0.15252854812398028, u'http://www.swaroopch.com/notes/Python'),
 (0.15252854812398028, u'http://coreblog.org/ats/making-app-engine-twitter-bot-in-15-lines-by-using-Flask'),
 (0.041598694942903712, u'http://www.kesiev/akihabara/'),
 (0.041598694942903712, u'http://www.greenteapress.com/thinkpython/'),
 (0.027732463295269141, u'http://www.kotono8.com/2010/04/23shokotan-twitter.html'),
 (0.027732463295269141, u'http://www.fontsquirrel.com/fontface/generator'),
 (0.027732463295269141, u'http://www.djangobook.com/')]
```

Which is a dictionary containing the 10 most similar url posts, that I haven’t tagged with the associated weighted score. And the results as can be seen by looking at the urls is pretty good seeing as how the j1z0 account only has a few links with are either django or learning python links.

So using a few simple algorithms we can build a link recommendation system for del.icio.us. This of course is barely scratching the surface of what can be done. But it is my hope that is will give you a brief idea of what can be done fairly easily with python and Collective Intelligence.

5. Conclusion

The world is changing and with more and more data present online the challenge of finding what one is looking for is becoming increasingly difficult. Those systems that help users do this in an effective manner will become highly prized. Luckily Collective Intelligence provides a good toolkit to make sense of the data, and python the ideal language to
implement those tools efficiently. Hopefully this short abstract and the accompanying presentation have provided a taste of what can be done and where to get started.

3. References