MULTI-DOMAIN SOCIAL RECOMMENDER SYSTEM

WEB&SOCIAL INFORMATION EXTRACTION REPORT

PROFESSOR: PAOLA VELARDI

JULY 25, 2018

Kadir Mert Ozcan 1780512

Contents

Introduction

Problem description

This work's aim is to make various experiments using the data from WikiMID dataset. The assignment is divided into five parts: collecting the categories of interests and assigning categories for each user; using these categories to detect communities in form of clusters; evaluating said clusters; collecting friendship information of given users with TwitterApi and clustering them; and finally, making recommendations to a set of users by selecting top three interest out of given six.

Tools Used

The project was developed using Python 3.6.2. The environment chosen was Jupyter Notebook that comes with Anaconda Navigator, due to its ability to create easy-to-understand structures with the appropriate use of markdown and code blocks.

List of users and their interests were obtained from WikiMID dataset.

Categories were collected using DBpedia resources.

Similarity matrices, clusters and evaluations were implemented with various sklearn methods.

Twitter friendship information was fetched using TwitterApi for Pyton.

Part 1 - Reading the Data and Categorizing the Interests

This project deals with 4 tab separated value (tsv) files that are included in WikiMID dataset. These files are 'friend-based_dataset.tsv', 'friend-based_interest_info.tsv', 'message-based_dataset.tsv', 'message-based_interest_info.tsv'.

'message-based_dataset' holds the data that associates users with a set of interests extracted from their messages. A row in this file looks like this:

1624748084	847672810965159938	2ENI	http://www.imdb.com/title/tt1219827

First and third columns are user and interest ids, respectively. Furthermore, 'message-based_dataset_info' stores the interest ids, and the part of the link to its Wikipedia page, as seen below:

2ENI	IMDb	WIKI:EN:Ghost_in_the_Shell_(2017_film)

Similarly, 'friend-based_dataset' contains the data extracted from the users' friendship list. A row in this file only consists of user id and interest id such as:

100000647	62513246

while 'friend-based_interest_info' contains interest ids and their Wikipedia page links:

62513246	WIKI:EN:J._K._Rowling

Collecting the data was a matter of reading through these documents and storing them in the memory. First order of business was to get the list of interests from "_info" files with their names:

```
#Dictionary to store <interestId, interestName> pairs
interestDict = {}
with open('friend-based_interest_info.tsv') as tsvfile:
  reader = csv.reader(tsvfile, delimiter="\t", quotechar='"')
  for row in reader:
    interestID = row[0]
    interestName = row[1][8:]
    interestDict[interestID] = interestName
with open('message-based_interest_info.tsv', encoding="utf8") as tsvfile:
  reader = csv.reader(tsvfile, delimiter="\t", quotechar='"')
  for row in reader:
    interestID = row[0]
    interestName = row[1][8:]
    interestDict[interestID] = interestName
```

FIGURE 1 - STORING THE INTEREST IDS AND THEIR NAMES

First 8 characters (WIKI:EN:) were cut down from the interest names as we only have English pages in our datasets.

More challenging part was to get the categories belonging to these interests, as this information didn't exist in the WikiMID dataset. A decision was made between using Wikipedia API and collecting these categories from an existing dataset. Using Wikipedia API would take a lot of time since it would require categories to be fetched for thousands of interest; but finding an up-to-date dataset was problematic as well.

In the end, I chose to use a DBPedia dataset named 'article_categories_en.nt'[1]. Some of the pages would be missing as the file dated back to April 2015, though hopefully this wouldn't cause much of a problem as the categories for most of the interests existed in the file.

```
<http://dbpedia.org/resource/Albedo> <http://purl.org/dc/terms/subject> <http://dbpedia.org/resource/Category:Climate_forcing> .
<http://dbpedia.org/resource/Albedo> <http://purl.org/dc/terms/subject> <http://dbpedia.org/resource/Category:Climatology> .
<http://dbpedia.org/resource/Albedo> <http://purl.org/dc/terms/subject> <http://dbpedia.org/resource/Category:Electromagnetic_radiation> .
<http://dbpedia.org/resource/Albedo> <http://purl.org/dc/terms/subject> <http://dbpedia.org/resource/Category:Radiometry> .
<http://dbpedia.org/resource/Albedo> <http://purl.org/dc/terms/subject> <http://dbpedia.org/resource/Category:Scattering,_absorption_and_radiative_transfer_(optics)> .
<http://dbpedia.org/resource/Albedo> <http://purl.org/dc/terms/subject> <http://dbpedia.org/resource/Category:Radiation> .
<http://dbpedia.org/resource/Anarchism> <http://purl.org/dc/terms/subject> <http://dbpedia.org/resource/Category:Anarchism> .
<http://dbpedia.org/resource/Anarchism> <http://purl.org/dc/terms/subject> <http://dbpedia.org/resource/Category:Political_culture> .
<http://dbpedia.org/resource/Anarchism> <http://purl.org/dc/terms/subject> <http://dbpedia.org/resource/Category:Political_ideologies> .
<http://dbpedia.org/resource/Anarchism> <http://purl.org/dc/terms/subject> <http://dbpedia.org/resource/Category:Social_theories> .
<http://dbpedia.org/resource/Anarchism> <http://purl.org/dc/terms/subject> <http://dbpedia.org/resource/Category:Anti-fascism> .
```

FIGURE 2 - ARTICLE_CATEGORIES_EN.NT

A dictionary object called interestCategoryDict was created in order to hold interest names and their corresponding set of categories. After this, article_categories_en.nt was crawled and interestCategoryDict rendered ready to be used in determining the dominant categories for each user.

[1] http://downloads.dbpedia.org/3.9/en/article_categories_en.nt.bz2

```
#List to store top categories for each user. Every entry is top n categories for a user.
categories = []
#Number of categories to get for each user
numberOfCats = 3
for user, interests in userInterestDict.items():
    #Dictionary to store category frequencies of all interest for a user
    localCategoryFreqDict = {}
    interestArr = interests.split(',')
    for interest in interestArr:
        if interestDict[interest] in interestCategoryDict:
            for category in interestCategoryDict[interestDict[interest]]:
                if category in localCategoryFreqDict:
                    localCategoryFreqDict[category] = localCategoryFreqDict[category] + 1
                else:
                    localCategoryFreqDict[category] = 1
    categories.append(" ".join(sorted(localCategoryFreqDict, key=localCategoryFreqDict.get, reverse=True)[:numberOfCats]
#print("--- %s seconds ---" % (time.time() - start_time))
```

FIGURE 3 - DERIVING PREFERRED INTEREST CATEGORIES

Final step for Part 1 was to derive a set of categories for each user. The method chosen for this was to take the 3 most common categories of each user's interests and store them in a list called "categories". Since the order of elements in a python list is persistent, storing this data in a dictionary as a <User, Categories> pair would be unnecessary. The set of users already exist as keys in the dictionary (userInterestDict) that was used to derive these set of categories. This means that, the first element in the list "categories" belongs to the user that is the first key in userInterestDict. One would be able to find out which categories belonged to which user by taking these two objects. Storing categories in a list also allowed me to use them as "documents" in the next step, clustering.

```
categories
```

```
['English_singer-songwriters NME_Awards_winners Grammy_Award-winning_artists',
 '21st-century_American_actresses Grammy_Award-winning_artists American_female_dancers',
 'United_Malays_National_Organisation_politicians Malaysian_politicians American_television_actors',
 'Fictional_assassins Fictional_telepaths Fictional_mechanics',
 "Parade_High_School_All-Americans_(boys'_basketball) African-American_basketball_players McDonald's_High_School_Al
l-Americans",
```

FIGURE 4 – PART OF THE CONTENTS OF 'CATEGORIES'

One thing to note is that, when choosing the categories, the ones that include 'living_people', 'deaths' and 'births' were ignored. Such categories are very frequent, yet it is not realistic to expect someone to be interested in these.

Part 2 – Clustering the Users

Clusters of users were generated using k-means clustering method. K-means algorithm select k objects (in this case it is chosen to be 30) as centers of clusters and assigns the nearest objects (objects with minimal distance to the center) to them. After that, new centers are selected in each cluster and all the elements are reassigned to new centers as in the first step. This is repeated until the algorithm converges.

Naturally, k-means algorithm can't accept our 'document' object as an input. To find the clusters, we must first create the tf-idf matrix. This was achieved using TfidfVectorizer from sklearn (Figure 5).

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.cluster import KMeans
from sklearn.metrics import adjusted_rand_score
vectorizer = TfidfVectorizer(tokenizer = my_tokenizer)
#X is a sparse term-document matrix of user-category pairs
X = vectorizer.fit_transform(categories)
```

FIGURE 5 - TFIDFVECTORIZER

A custom tokenizer was defined to prevent Vectorizer from tokenizing categories such as 'Warhammer_400,000_Species' as 'Warhammer_400' and '000_Species'. Thanks to this tokenizer (Figure 6), every category counts as a single token.

```
def my_tokenizer(s):
    return s.split(' ')
```

FIGURE 6 - CUSTOM TOKENIZER

fit_transform method produces a 444744 x 23863 sparse matrix of documents (users) and terms (categories). With such high dimensional space, it is very unlikely for K-means to give us meaningful clusters. This phenomenon is called curse of dimensionality. Indeed, during my first trials, the resulting clusters were giving very low scores upon evaluation, as low as 0.003. This is the result of overlapping clusters. Solution to this was reducing dimensionality. However, using Principal Component Analysis would require me to convert the sparse matrix into a dense one, hence causing memory error.

Enter Truncated SVD. (Figure 7) Instead of doing mean normalization like PCA, Truncated SVD performs linear dimensionality reduction by means of truncated singular value decomposition.

```
from sklearn.decomposition import TruncatedSVD
from sklearn.random_projection import sparse_random_matrix
svd = TruncatedSVD(n_components=8, n_iter=10, random_state=42)
reducedX = svd.fit_transform(X)
```

FIGURE 7 - TRUNCATED SVD

fit_transform method returns the reduced matrix, which will be used on our k-means model (Figure 8)

```
kmeans = KMeans(n_clusters=30, init='k-means++',n_init=10, verbose=0)
kmeans.fit(reducedX)
```

FIGURE 8 - K-MEANS CLUSTERING

Part 3 - Evaluating the Clusters

Since the ground truth labels are not know, it is not possible to evaluate the clusters using methods such as Adjusted Rand index or Fowlkes-Mallows scores. Evaluation must be performed using the model itself. This issue can be addressed by computing Silhouette Coefficient of all samples. (Figure 9)

```
from sklearn import metrics
metrics.silhouette_score(reducedX, kmeans.labels_, metric='euclidean', sample_size = 1000)
```

```
0.79683111782414362
```

FIGURE 9 - CALCULATING SILHOUETTE SCORE

Silhouette Coefficient for a single sample is given as:

$$s = \frac{b - a}{\max(a, b)}$$

Where **a** is the distance between a sample and all other points in the same class, whereas **b** is the mean distance between a sample and all other points in the nearest cluster. Scores range between -1 and 1. Scores closer to 0 mean there are overlapping clusters; the problem that was solved in this project by using TruncatedSVD.

Using sklearn's silhouette_score algorithm on our clusters gave us score of 0.796. Increasing the number of clusters or decreasing the number of categories for each user can yield higher scores.

Part 4 - Clustering Twitter Users

We are given a set of users to fetch their friendship information using Twitter Api. For this, we need a set of keys provided by twitter. As of July 2018, twitter reviews every application for a development account; this process might take some time. Fortunately, I applied for a development account at the beginning of this course.

```
from twitter import *

CONSUMER_KEY = '9HqYMGHtCok5QINP6n4YokR9U'
CONSUMER_SECRET = 'jfOG4iLpX8pKgZ4ZGbjdPzCxZjlI6VNUfY1RKxrYSgPoDHzRzF'
ACCESS_TOKEN = '973238402580664325-WiXwkPnQ5YzGrwTv0nr9omsjspfxyhT'
ACCESS_TOKEN_SECRET = '4z4q0zoJ7Pfqv03IiQeGRQDmX2BM5o1FsbkeXipCyaLKA'

twitter = Twitter(auth = OAuth(ACCESS_TOKEN,
                  ACCESS_TOKEN_SECRET,
                  CONSUMER_KEY,
                  CONSUMER_SECRET))
```

FIGURE 10 - TWITTER TOKENS

```
#start_time = time.time()
#Dictionary to store <userId, friends> pair
twitterUserList = {}
with open('S21.tsv') as tsvfile:
  reader = csv.reader(tsvfile, quotechar='"')
  for row in reader:
    twitterUserList[row[0]] = ""
    try:
        friends = twitter.friends.ids(user_id=row[0])['ids']
        twitterUserList[row[0]] = ' '.join(str(x) for x in friends)
    except Exception as e:
        del twitterUserList[row[0]]
        print(e)
    time.sleep(60)
#print("--- %s seconds ---" % (time.time() - start_time))
```

FIGURE 11 - FETCHING FRIENDSHIP INFORMATION

The list of users is read from the file provided ('S21'), and stored in the dictionary "twitterUserList" as <User, friends> pair. (Figure 11)

To cluster these users, the similarity between them are calculated using cosine similarity method (Figure 12) and fed into the k-means algorithm same as the users in Part 2.

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.cluster import KMeans
from scipy.spatial.distance import pdist, squareform

vectorizer = CountVectorizer()
X = vectorizer.fit_transform(list(twitterUserList.values()))
#Similarity Matrix
cs_users = squareform(pdist(X.toarray(), 'cosine'))
```

Clusters are created with the same method used for Part 2

```
from sklearn.decomposition import TruncatedSVD
from sklearn.random_projection import sparse_random_matrix
svd = TruncatedSVD(n_components=5, n_iter=7, random_state=42)
reducedX = svd.fit_transform(cs_users)
kmeans = KMeans(n_clusters=5, init='k-means++',n_init=1000, verbose=0)
kmeans.fit(reducedX)
```

FIGURE 12 - CLUSTERING TWITTER USERS

Cosine similarity is calculated after the friend lists are vectorized using CountVectorizer by converting a collection of text documents to a matrix of token counts. Similar to how the categories were tokenized, every friendId counts as a token. Using pdist method, we obtain the pairwise distance between observations as one-dimensional array using the following formula:

$$cos(\theta) = \frac{A.B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} {A_i}^2}\sqrt{\sum_{i=1}^{n} {B_i}^2}}$$

squareform converts it to a distance matrix, ready to be used for k-means clustering.

Disadvantage of using this method is that, similarities are calculated without taking global structure into account. A better way to calculate similarity between users with their list of friends would be using the *SimRank* method. *SimRank* measures the similarity of the structural context in which objects occur based on their relationship with other objects; not just the object it compares to. Two objects are considered to be similar if they are referenced by similar objects.

There aren't any Python libraries with SimRank. Figure 13 shows an adaptation of a SimRank that was found on github[2]. The algorithm very slowly though, and it may be possible to get a memory error if the number of tokens (unique friends) are too large. Nevertheless, I felt a need to include this in the project as I believe this is a better way to measure the similarity between two users.

```
user_sim = matrix(np.identity(len(users)))
friend_sim = matrix(np.identity(len(friends)))

def get_friends_num(user):
    q_i = users.index(user)
    return graph[q_i]

def get_friends(user):
    series = get_friends_num(user).tolist()[0]
    return [ friends[x] for x in range(len(series)) if series[x] > 0 ]

def user_simrank(q1, q2, C):
    if q1 == q2 : return 1
    prefix = C / (get_friends_num(q1).sum() * get_friends_num(q2).sum())
    postfix = 0
    for friend_i in get_friends(q1):
        for friend_j in get_friends(q2):
            i = friends.index(friend_i)
            j = friends.index(friend_j)
            postfix += friend_sim[i, j]

    return prefix * postfix


def simrank(C=0.8):
    global user_sim, friend_sim

    # users simrank
    new_user_sim = matrix(np.identity(len(users)))
    for qi in users:
        for qj in users:
            i = users.index(qi)
            j = users.index(qj)
            new_user_sim[i,j] = user_simrank(qi, qj, C)
    user_sim = new_user_sim
    return user_sim
```

FIGURE 13 - SIMRANK

NOTE: Unfortunately, twitter has a rate limit for fetching the data. Only 15 requests are allowed in 15 minutes. To overcome this, the "for" loop used in the project needs to be staggered to request only one friendship information per minute. This was done by using "sleep(60)" between each request. The file provided (S21.tsv) has 1500 users, this makes 1500 minutes

[2] https://github.com/littleq0903/simrank

which means one needs to wait for nearly 24 hours to fetch all the data. For this reason, the final clustering was done on a small subset of users. Nonetheless, clustering algorithm also works on all the users.

Part 5 – Recommending Interests

Aim of this part is to find the most important 3 topics out of given 6 for a set of users. To do this all the categories of a user's existing interests are counted and stored in a dictionary called userCategoryFrequency. The dictionary's structure is <User, <Category, Frequency>>. (Figure 14)

```
#Dictionary to store <User, <Category, Frequency>> pairs.
userCategoryFrequency = {}
for user, interestList in userInterestDict2.items():
    #category frequencies for each user, will be added to userCategoryFrequency as a value
    categoryFrequencies = {}
    for interest in interestList:
        if interest in interestCategoryDict2:
            categories = interestCategoryDict2[interest]
            for category in categories:
                if category in categoryFrequencies:
                    categoryFrequencies[category] = categoryFrequencies[category] + 1
                else:
                    categoryFrequencies[category] = 1
    userCategoryFrequency[user] = categoryFrequencies
```

FIGURE 14 - COUNTING THE CATEGORIES

Then, we calculate the total frequencies of the categories of interests to be proposed, find the top three that has the most common categories in userCategoryFrequency. These three interests are added to the top3Recommendations dictionary as <UserId, InterestList> pairs. (Figure 15)

```
#Dictionary that holds <UserId, Interest> pairs for top 3 recommendations
top3Recommendations = {}
for user, interestList in recommendDict.items():
    #Dictionary that holds an Interest-Frequency pair. Frequency is the total frequency of categories for that interest
    localCategoryFrequencyDict = {}
    for interest in interestList:
        frequency = 0
        for category in interestCategoryDict2[interest]:
            if category in userCategoryFrequency[user]:
                frequency = frequency + userCategoryFrequency[user][category]
        localCategoryFrequencyDict[interest] = frequency
    #get top 3 interest with most common categories
    topInterests = list(dict(sorted(localCategoryFrequencyDict.items(), key=lambda x: x[1], reverse=True)[:3]).keys())
    top3Recommendations[user] = topInterests
```

FIGURE 15 - TOP 3 RECOMMENDATIONS

Conclusion

In this work, I used python and sklearn algorithms to find solutions for the given 5 problems. Despite dealing with the large amount of data, with the efficient use of dictionaries, no memory overflows were observed.

Future Imporovements

Categorizing the interests was the trickiest part of the project, and in my opinion could have been done better. Source of the interest could have been considered and weights could have been given to certain categories. Sadly, my trials on giving weights to categories all resulted in having worse clusters.

In addition to this, a better implementation of SimRank could have been used to cluster users based on their friend lists.