Project of Web and Social Information Extraction

Alessi Emanuele
1486470

Ferrante Ivan
1563390

Introduction

For our project, in order to make the various results viewable with plots, we have chosen to use Python jupyter notebooks. In addition, the jupyter notebook allowed us to create this pdf file in which we can explain step by step our code. Our work is organized in 3 notebook files:

- Wikipedia Cluster
- Twitter Cluster
- Recommender System

Furthermore, there are also two python files in which we have written some functions that we used in notebook files:

- **crawler.py:** contains some functions used in cluster analysis of Twitter and Wikipedia
- **recommender_system.py:** contains some function used in our recommender system

All the data genereted by our project are stored in the DATA folder, except for some json file that would have been too heavy to send.

Below, going forward with reading of this pdf, there are all the code explained and the results commented for all steps of our project.

1. Wikipedia Cluster

1.1 Wikipedia Clusters

In this notebook we have done all the tasks required from the step 1,2 and 3 of the project. From scraping the data to analysis of the obtained clusters. We have used BeautifulSoup as library for scraping data, and sklearn as library for data analysis.

```
In [1]: import urllib.request as ul
        from bs4 import BeautifulSoup
        from tqdm import tqdm
        import pickle
        import os
        import numpy as np
        from IPython.display import display, clear_output
        from sklearn.cluster import KMeans
        from sklearn.decomposition import PCA
        import matplotlib.pyplot as plt
        from sklearn.metrics import silhouette_score, silhouette_samples
        import matplotlib.cm as cm
```

We can start with scraping wikipedia pages. Initially we designed a crawler using the sequential approach, but in this way we noticed that it takes about 12 hours to finish the computation. So, we have implemented a **parallelized** version of this crawler that can complete this task in 1 hour, the respective code is in the file **crawler.py** .

In the scraper.py file there are two function:

- **parse(line):** Is a parallelized function used for parsing wikipedia pages, in order to retrieve the respective categories. We used find(...) function of BeautifulSoup to identify the categories in the html schema, then we return a couple composed by (user_ID, list of category)

- **one_hot_encoding(set, size):** Is a function used to obtain an one hot vector for each user mapped with categories of his wikipedia pages.

1.2 Categories reduction:

Wikipedia is a not structured source of information, so given a subcategory of one wikipedia page, if we try to find a root for that subcategory, this is an impossible task. Then, we have implemented an algorithm that reduce the number of category and retrieve the most significant categories for each page. In details, when we have retrieved all the categories for one wikipedia page, we split all the words of this categories, remove all the stopwords, and count the remains words. Then, for each wikipedia page we store the **3 most repeated words**. In this way we try to extract a contest from each Wikipedia page.

At the end of the execution of crawler.py, we create two python dictionary: categories and preferencies.

- **categories:** is a dictionary that has name of category (string) as key and unique id (integer) as value.

- **preferencies:** is a dictionary that has user_id (integer) as key and list of categories (list of strings) as value that contains all the categories associated to user's preferencies.

At the end of the crawling (done by crawler.py), we store this two dictionaries in two pickle files: **categories.pkl**, **preferencies.pkl**

At this point we perform the PCA algorithm to reduce dimension of our data in order to make our data visualizable on the plot.

```
In [2]: f = pickle.load(open('preferencies_wiki_final.pkl', 'rb'))
        dataset = np.array([f[user] for user in f])
```

```
In [4]: pca = PCA(n_components=2).fit(dataset)
        pca_2d = pca.transform(dataset)
```

```
In [4]: pca_2d.shape
```

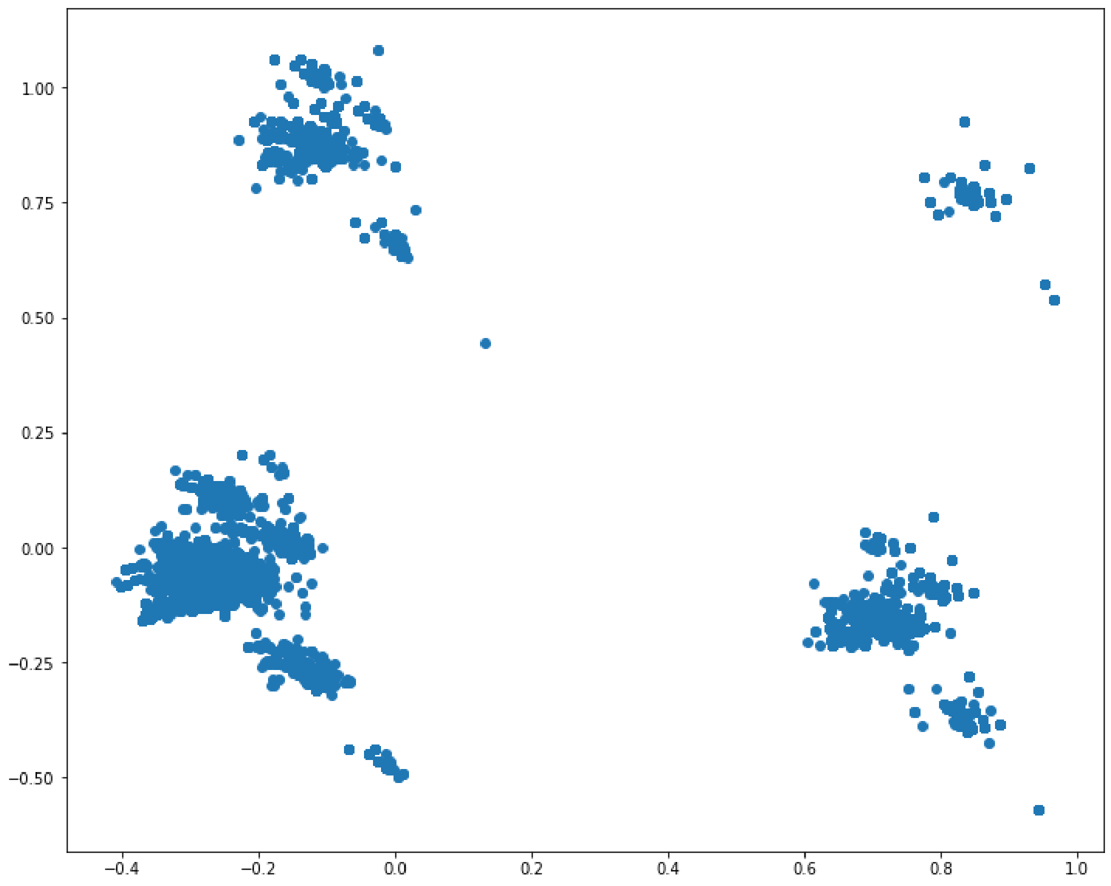
```
Out[4]: (58789, 2)
```

```
In [5]: pca_2d[:20]
```

```
Out[5]: array([[-0.2527906 , -0.06948457],
               [-0.34559589, -0.12020253],
               [-0.28146307,  0.09834449],
               ...,
               [-0.26566972, -0.07716841],
               [ 0.82399842, -0.08819858],
               [-0.24238285, -0.07134737]])
```

Above the plot of PCA results, before that we try to find clusters:

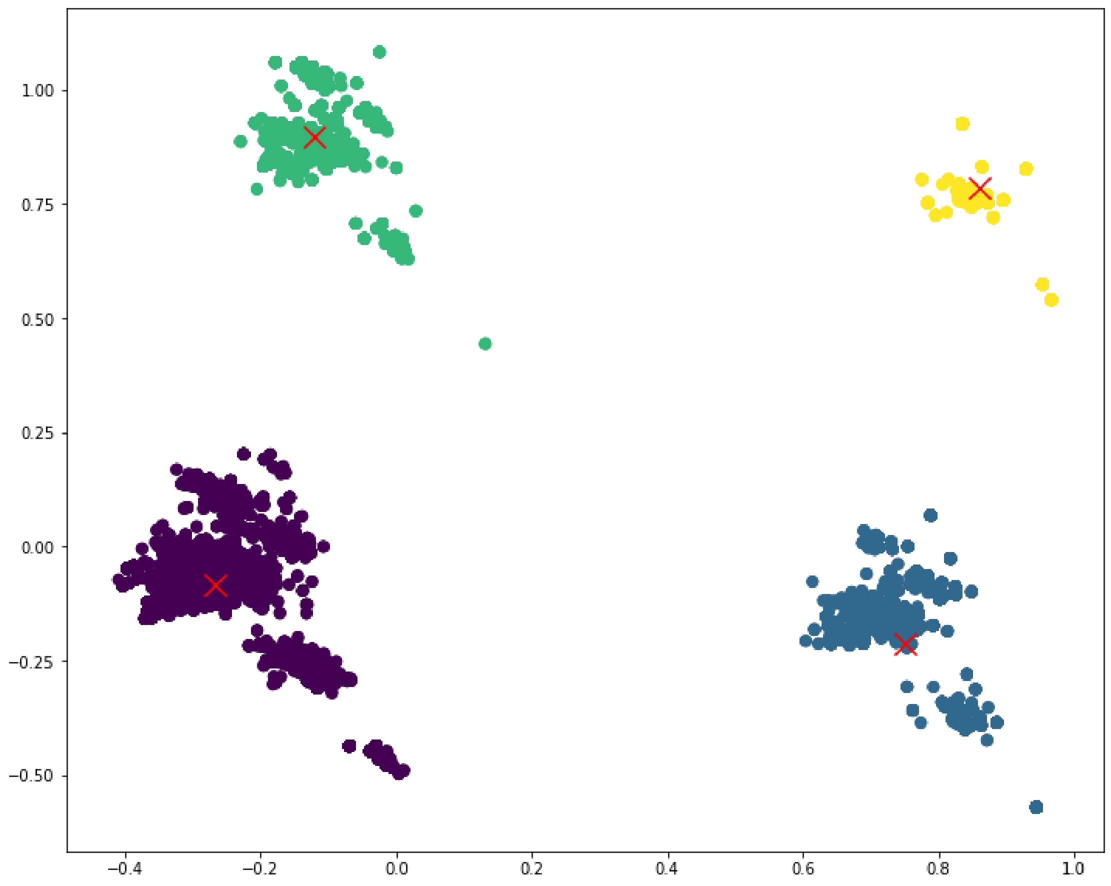
```
In [6]: plt.figure(figsize=(12,10))
        plt.scatter(pca_2d[:, 0], pca_2d[:, 1])
        plt.show()
```



1.3 Clusters Analysis

The PCA plot suggests us to try with 4 clusters, and this is the result obtained. At each cluster we have assigned one color, and we denote with **red X** the centroid of each cluster

```
In [5]: kmeans = KMeans(n_clusters=4).fit(pca_2d)
        y_kmeans = kmeans.predict(pca_2d)
        plt.figure(figsize=(12, 10))
        plt.scatter(pca_2d[:, 0], pca_2d[:, 1], c=y_kmeans, s=50)
        centroids = kmeans.cluster_centers_
        plt.scatter(pca_2d[:, 0], pca_2d[:, 1], c=y_kmeans, s=50);
        plt.scatter(centroids[:, 0], centroids[:, 1], s=200, marker='x', c='red')
        plt.show()
```



In order to evaluate the clusters we have choose to use **Silhouette score** as metrics. For each point i, first find the average distance A_i between i and all other points in the same cluster, that is a measure of cohesion. Then, find the average distance B_i between i and all points in the nearest cluster, that is a measure of separation from the closest other cluster. The silhouette coefficient for s_i is defined as the difference between B and A divided by the greater of the two $max_{A,B}$.

$s_i = \frac{B_i - A_i}{max_{A_i,B_i}}$

Since the cost of computation for this score is very high because we have 58000 elements in our dataset, we can retrieve this score for cluster analysis with a sample size of 100:

```
In [4]: pca = PCA(n_components=2).fit(dataset)
        pca_2d = pca.transform(dataset)
        kmeans = KMeans(n_clusters=4).fit(pca_2d)
        y_kmeans = kmeans.predict(pca_2d)

        n_clusters = 4
        silhouette_avg = silhouette_score(pca_2d, y_kmeans, sample_size=100)

        print("For n_clusters =", n_clusters,
              "The average silhouette_score is :", silhouette_avg)
```

```
For n_clusters = 4 The average silhouette_score is : 0.9062060323202136
```

2 Twitter Cluster

2.1 Twitter Cluster

In this notebook file we had implemented a clusters analysis on a Twitter's profiles related of the file S21.tsv. After a careful evaluation of the various library that Python offers for our scope, we had choice the official API of Twitter in order to get the various information on twitter profiles.

```
In [1]: import twitter
        import pandas as pd
        import json
        import datetime
        import csv
        import numpy as np
        from sklearn.decomposition import PCA
        import matplotlib.pyplot as plt
        from sklearn.cluster import KMeans
        from sklearn.cluster import KMeans

        api = twitter.Api(consumer_key='2Nn22OkPPZaBoarCjxcP7Tr70',
                          consumer_secret='9VdkWxOFvmrnjEzbG3SoMkh9kPKB5ToHb6RKnwGyso8XrAlPk2'
                          access_token_key='204761432-jUj8moOZ6ccKo7s48ufl8mpPB3amryzzvidYIhDA
                          access_token_secret='2XVFSENxf7RNqwysvHYsSh1j5itKpdItJtA6khjxd7Dxp',
                          sleep_on_rate_limit = True )

        api.VerifyCredentials()
```

```
Out[1]: User(ID=204761432, ScreenName=Ivan_FCT)
```

2.2 Crawling Twitter Friendship Data

We started with an analysis based on friendship information. For each twitter's profile we get, throught the functions provided to Twitter API (getFriendsID(...) and getFollowersID(...)), we retrive the list of IDs that all our users follow and the list of IDs that follow our users. This computation is very slow because Twitter API allow to retrive this information for 15 user every 15 minutes. So for each iteration we print summary information of the data obtained in order to try to understand at what point is the execution of the process.

```
In [ ]: with open("S21.tsv") as f:
            reader = csv.reader(f)
```

```
        s21 = []
        for row in reader:
            s21.append(int(row[0]))

    count = 0
    data = {}


    for id_user in s21:
        if id_user in data.keys():
            continue
        else:
            try:
                data[id_user] = api.GetFriendIDs(user_id=id_user)
                tot = len(data[id_user])
                count += 1
                with open("friends_id.json", "w") as write_file:
                    json.dump(data, write_file)
                print(id_user , ' ----> DONE! Nř:', count, '- TOT_FRIENDS:', tot,
                      '- WHEN: ', datetime.datetime.now().time())
                write_file.close()

            except Exception as err:
                print(err)

    for id_user in s21:
        if id_user in data.keys():
            continue
        else:
            try:
                data[id_user] = api.GetFollowerIDs(user_id=id_user)
                tot = len(data[id_user])
                count += 1
                with open("followers_id.json", "w") as write_file:
                    json.dump(data, write_file)
                print(id_user , ' ----> DONE! Nř:', count, '- TOT_FOLLOWERS:', tot,
                      '- WHEN: ', datetime.datetime.now().time())
                write_file.close()

            except Exception as err:
                print(err)
```

After 30 hours of execution we have our list of friends and followers that we have stored in JSON file. At this point we create the intersection of this two sets, in order to obtain only the IDs that follow our users and that our users following too.

```
In [14]: with open("friends_id.json", "r") as read_file:
             friends = json.load(read_file)
```

```
        print('Tot friend_id keys:',len(friends.keys()))

        with open("followers_id.json", "r") as read_file:
            follows = json.load(read_file)

        print('Tot followers_id keys:',len(follows.keys()))

        keys_a = set(friends.keys())
        keys_b = set(follows.keys())
        intersection = keys_a & keys_b

        print('Tot keys in comune trovate:',len(intersection))

        result = {}
        tot_follow = []
        for _id in intersection:
            result[_id] = set(friends[_id]) & set(follows[_id])

        for lista in result.values():
            for v in lista:
                tot_follow.append(v)

        print('Totale user trovati:', len(set(tot_follow)))
```

```
Tot friend_id keys: 1420
Tot followers_id keys: 1433
Tot keys in comune trovate: 1411
Totale user trovati: 760810
```

Now, at each ID of our intersection set we associate a number that we will use in the one hot encoding function below. So user mapper's length is the same length of our intersection sets.

```
In [3]: user_mapper = {}
        c = 0

        for user in result:
            for follower in result[user]:
                if follower not in user_mapper:
                    user_mapper[str(follower)] = c
                    c += 1
```

```
In [4]: len(user_mapper)
```

```
Out[4]: 760810
```

The function one_hot_encoding take as parameter the list of intersection between followers/friends and the user mapper and create one hot vector for each user in S21 file, that is a vector with length of 760810 that have only zeros or 1.

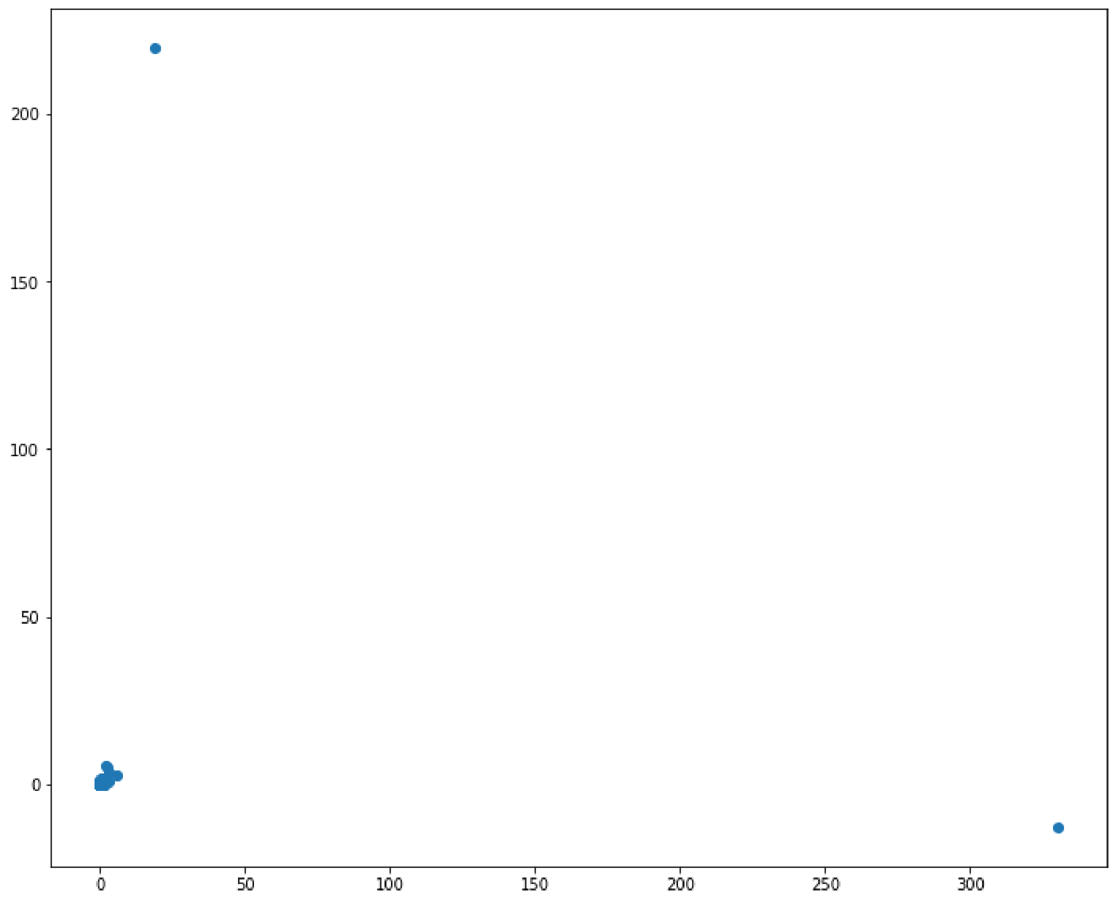
```
In [2]: def one_hot_encoding(follower_list, mapper):
            v = np.zeros(len(mapper))
            for follower in follower_list:
                try:
                    position = mapper[str(follower)]
                    v[position] = 1
                except:
                    pass
            return v

        dataset = [one_hot_encoding(result[user], user_mapper) for user in result]
        len(dataset[1])
```

In order to make our results viewable through plot we use PCA to obrain for each vector a couple of number.

```
In [15]: del result
         pca = PCA(n_components=2).fit(dataset)
         pca_2d = pca.transform(dataset)

In [18]: pca_2d
```

Our friendship information are not informative enough, this may be due to the few friendships our users have in common, in fact we can see how all the users, represented by points in the plot below, are concentrated in the same area. So we can change our strategy!

```
In [23]: plt.figure(figsize=(12,10))
         plt.scatter(pca_2d[:, 0], pca_2d[:, 1])
         plt.show()
```



```
In [21]: kmeans = KMeans(n_clusters=2).fit(pca_2d)
         y_kmeans = kmeans.predict(pca_2d)
```

2.3 Crawling Twitter Hashtag Data

We change our strategy: now we want explore the last hashtag used by our users given by the S21.tsv file! This time we will use the function named GetUserTimeline(...) with parameter count=100, that give us the list of last 100 tweet for each user. Starting from this tweets we can retrieve the hashtag that each user have used.

```
In [ ]: with open("S21.tsv") as f:
            reader = csv.reader(f)
            s21 = []
            for row in reader:
                s21.append(int(row[0]))

        count = 0
        data = {}
        for id_user in s21:
            if id_user in data.keys():
                continue
            else:
                try:
                    t = api.GetUserTimeline(user_id=id_user, count=100)
                    tweets = [i.AsDict() for i in t]
                    data[id_user] = [tweet['hashtags'][0]['text'] for tweet in tweets if tweet
                    tot = len(data[id_user])
                    count += 1
                    with open("hashtags.json", "w") as write_file:
                        json.dump(data, write_file)
                    print(id_user , ' ----> DONE! Nř:', count, '- TOT_HASHTAG:', tot,
                          '- WHEN: ', datetime.datetime.now().time())
                    write_file.close()

                except Exception as err:
                    print(err)
```

As before, we stored all the hashtag retrieved in a JSON file, then we map again all the hashtags and finally we create our second dataset with one hot vector for each user mapped on his hashtags.

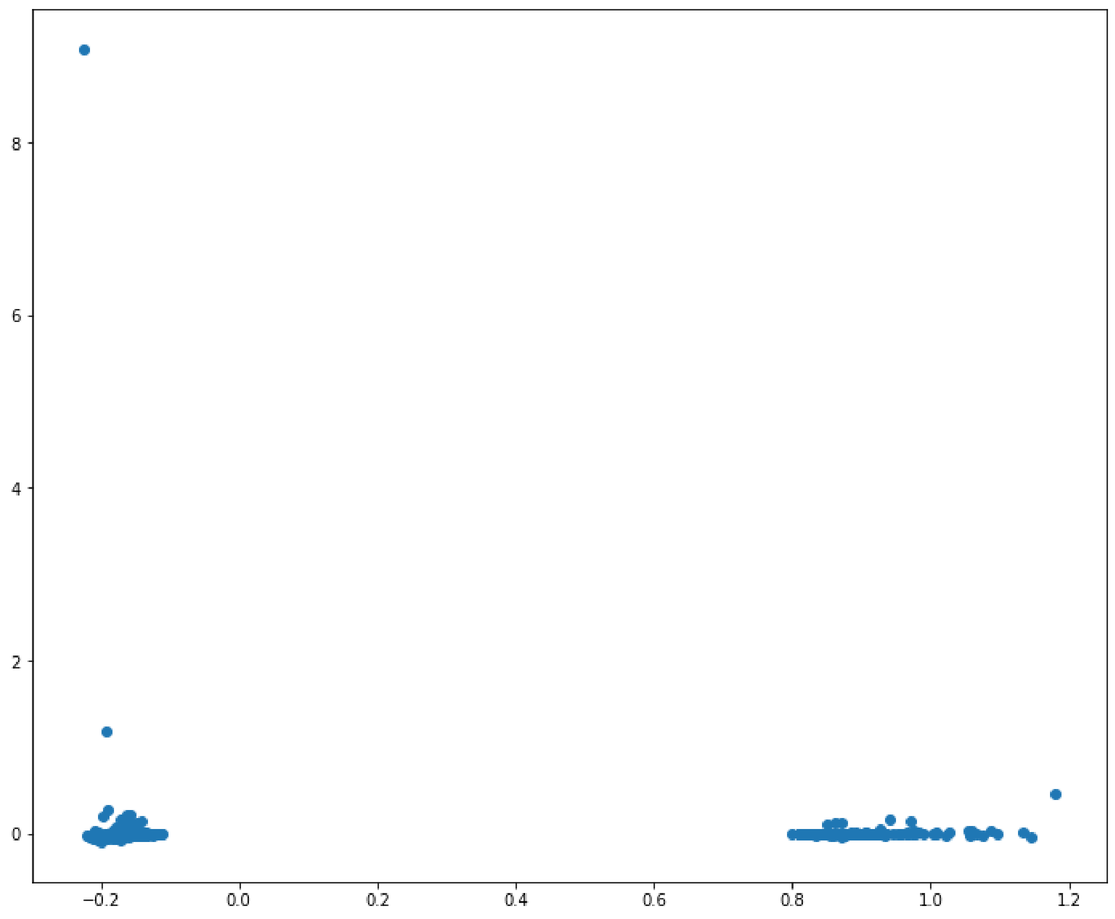
```
In [4]: with open("hashtags.json", "r") as read_file:
            hashtags = json.load(read_file)

        tot_hash = []
        for lista in hashtags.values():
            for v in lista:
                tot_hash.append(v)

        print('Totale hashtag trovati:', len(set(tot_hash)))

        h = 0
        hash_mapper = {}
        for hashtg in tot_hash:
            if hashtg not in hash_mapper:
                hash_mapper[str(hashtg)] = h
                h += 1

        print('Numero chiavi hash_mapper:', len(hash_mapper.keys()))

        dataset_hash = [one_hot_encoding(hashtags[hashtg], hash_mapper) for hashtg in hashtags
        print('---> dataset_hash sarà una matrice', len(dataset_hash), 'x', len(dataset_hash[0
```

```
Totale hashtag trovati: 11419
```

```
Numero chiavi hash_mapper: 11419
---> dataset_hash sarà una matrice 1422 x 11419
```

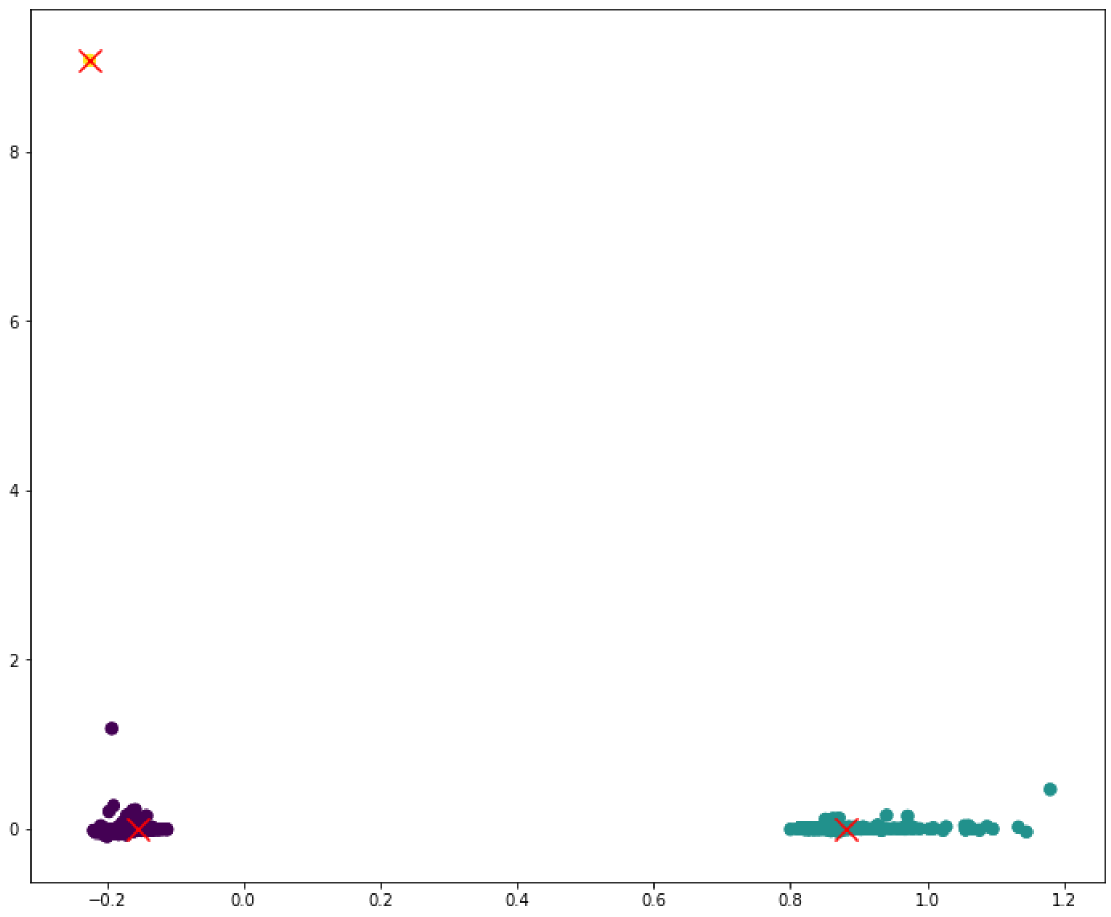
Again we compute PCA analysis in order to obtain results that we can plot

```
In [5]: pca = PCA(n_components=2).fit(dataset_hash)
        pca_2d = pca.transform(dataset_hash)

In [6]: pca_2d.shape

Out[6]: (1422, 2)

In [7]: pca_2d

Out[7]: array([[ 0.88422247, -0.00285513],
               [-0.14671981, -0.00672723],
               [-0.15131576, -0.00716176],
               ...,
               [-0.15712744, -0.00809285],
               [-0.14671981, -0.00672723],
               [-0.11236289, -0.01152587]])

In [8]: plt.figure(figsize=(12,10))
        plt.scatter(pca_2d[:, 0], pca_2d[:, 1])
        plt.show()
```



Using hashtags as a comparison item between users we obtain a good result. All the users are clustered in 2 macro area where the red X are the centroids of the cluster and only few users are clustered in another point (yellow cluster), this may refers to users with only 1 hashtag found.

```
In [9]: kmeans = KMeans(n_clusters=3).fit(pca_2d)
        y_kmeans = kmeans.predict(pca_2d)
        plt.figure(figsize=(12, 10))
        plt.scatter(pca_2d[:, 0], pca_2d[:, 1], c=y_kmeans, s=50)
        centroids = kmeans.cluster_centers_
        plt.scatter(pca_2d[:, 0], pca_2d[:, 1], c=y_kmeans, s=50);
        plt.scatter(centroids[:, 0], centroids[:, 1], s=200, marker='x', c='red')
        plt.show()
```



8
6
4
2
0
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2

3. Recommendation System

3.1 Recommender System

For our Recommender System we have write more function stored in the file reccomander_system.py. We have chosen to implement more scoring metrics in order to evaluate the differences between them. The logic of this functions is reported below:

- **init(self, preferencies, categories, s23_file)** : This function is used to instantiate an reccomender system object.
- **cosine_similarity(self, v, w):**: This function define one of our metrics of scoring. Cosine similarity assign to each user's wikipedia page a score based on his preferences.
- **pearson_correlation(self, v, w):**: This function define one of our metrics of scoring. Pearson correlation computes the linear correlation between two variables that, in out case, are two users preferencies vector. It has a value between +1 and 1, where 1 is total positive linear correlation, 0 is no linear correlation, and 1 is total negative linear correlation.
- **euclidean_distance(self, v, w):**: This function define one of our metrics of scoring. Euclidean distance or Euclidean metric compute the straight-line distance between two points in Euclidean space.
- **most_relevant_pages(self, user_id, metrics):** This function take in input a user_id and return the 6 page ordered by score computed by one of three metrics available.

```
In [1]: from recommender_system import RecommenderSystem
        import pickle
        from itertools import islice
```

In the pickle categories we have stored the mapping between one category and its ID.

```
In [3]: categories = pickle.load(open('DATA/categories.pkl', 'rb'))
        first20pairs = {k: categories[k] for k in list(categories)[:20]}
        first20pairs

Out[3]: {'1981 births': 0,
         '1987 births': 10,
         '20th-century Japanese actresses': 3,
         '21st-century Japanese actresses': 4,
         'Actresses from Kanagawa Prefecture': 2,
         'Ambassadors of supra-national bodies': 11,
         'Australian Open (tennis) champions': 12,
         'Expatriate sportspeople in the United States': 13,
         'French Open champions': 14,
         "Grand Slam (tennis) champions in women's singles": 15,
         'Guggenheim Fellows': 8,
         'Japanese film actresses': 5,
         'Japanese stage actresses': 6,
         'Japanese television actresses': 7,
         'Living people': 1,
         'Maria Sharapova': 9,
         'Olympic medalists in tennis': 16,
         'Olympic silver medalists for Russia': 17,
         'Olympic tennis players of Russia': 18,
         'Sportspeople from Bradenton, Florida': 19}
```

In the pickle preferencies we assign to each user a one hot vector that contains 1 in the positions of the page that he likes, 0 otherwise. On this vector is computed the cosine similarity.

```
In [4]: preferencies = pickle.load(open('DATA/preferencies.pkl', 'rb'))
        first20pairs = {k: preferencies[k] for k in list(preferencies)[:20]}
        first20pairs

Out[4]: {'100618369': array([1, 1, 0, ..., 0, 0, 0], dtype=int32),
         '101684764': array([1, 1, 0, ..., 0, 0, 0], dtype=int32),
         '1018670268': array([1, 1, 0, ..., 0, 0, 0], dtype=int32),
         '101935414': array([1, 1, 1, ..., 0, 0, 0], dtype=int32),
         '101948722': array([1, 1, 0, ..., 0, 0, 0], dtype=int32),
         '1025469163': array([1, 1, 0, ..., 0, 0, 0], dtype=int32),
         '103232008': array([0, 1, 0, ..., 0, 0, 0], dtype=int32),
         '103461258': array([0, 1, 0, ..., 0, 0, 0], dtype=int32),
         '1040719962': array([0, 1, 0, ..., 0, 0, 0], dtype=int32),
         '104239528': array([1, 1, 0, ..., 0, 0, 0], dtype=int32),
         '104265805': array([0, 1, 0, ..., 0, 0, 0], dtype=int32),
         '104431004': array([1, 1, 0, ..., 0, 0, 0], dtype=int32),
         '105143827': array([0, 1, 0, ..., 0, 0, 0], dtype=int32),
         '105150125': array([0, 1, 0, ..., 0, 0, 0], dtype=int32),
         '105320169': array([0, 1, 0, ..., 0, 0, 0], dtype=int32),
         '105553307': array([1, 1, 0, ..., 0, 0, 0], dtype=int32),
         '105567943': array([1, 1, 0, ..., 0, 0, 0], dtype=int32),
         '10569722': array([1, 1, 0, ..., 0, 0, 0], dtype=int32),
         '1057516135': array([0, 1, 0, ..., 0, 0, 0], dtype=int32),
         '1058041224': array([0, 1, 0, ..., 0, 0, 0], dtype=int32)}
```

Then, we can instantiate a Recommender System object, and we can use the function **most_relevant_pages** in order to retrieve the sorted ranking of the 6 page proposed to each user in the file S23.tsv.

```
In [5]: rs = RecommenderSystem(preferencies, categories, open('DATA/S23.tsv', 'r'))
```

Below some example of our recommender system that suggest at each user the 3 pages with scores of all matrics based on his interest:

```
In [6]: rs.most_relevant_pages('101684764', metrics = 'all')

Top 3 (cosine similarity):  ['Heather_Headley', 'Jennifer_Lopez', 'Ryan_Seacrest']
Top 3 (euclidean distance):  ['Jennifer_Lopez', 'Heather_Headley', 'Maxene_Magalona']
Top 3 (pearson correlation):  ['Heather_Headley', 'Jennifer_Lopez', 'Ryan_Seacrest']
```

But if we want we can check all the scores for each metrics:

```
In [10]: rs.most_relevant_pages('101684764', metrics = 'cosine')

Out[10]: [(0.12893946815413682, 'Heather_Headley'),
          (0.11877671491968554, 'Jennifer_Lopez'),
          (0.11310517053068245, 'Ryan_Seacrest'),
          (0.10735733858321811, 'Maxene_Magalona'),
          (0.09552948218914535, 'Julian_Assange'),
          (0.09273288940232355, 'ITV_News_Anglia')]

In [11]: rs.most_relevant_pages('101684764', metrics = 'pearson')

Out[11]: [(0.12071388481136608, 'Heather_Headley'),
          (0.1143576980106862, 'Jennifer_Lopez'),
          (0.10512785429755785, 'Ryan_Seacrest'),
          (0.09974802024396087, 'Maxene_Magalona'),
          (0.0884025629236834, 'Julian_Assange'),
          (0.08525707535796076, 'ITV_News_Anglia')]

In [12]: rs.most_relevant_pages('101684764', metrics = 'euclidean')

Out[12]: [(48.33218389437829, 'Jennifer_Lopez'),
          (48.569537778323564, 'Heather_Headley'),
          (48.67237409455183, 'Maxene_Magalona'),
          (48.67237409455183, 'Ryan_Seacrest'),
          (48.703182647543684, 'Julian_Assange'),
          (48.76474136094644, 'ITV_News_Anglia')]
```

Below there are some examples on other users of how our recommender system works:

```
In [13]: rs.most_relevant_pages('104431004', metrics = 'all')

Top 3 (cosine similarity):  ['The_Beatles', 'Miki_Nadal', 'La_Terra']
Top 3 (euclidean distance):  ['Berto_Romero', 'Damien_Rice', 'Chris_Boswell']
Top 3 (pearson correlation):  ['The_Beatles', 'Miki_Nadal', 'La_Terra']
```

```
In [14]: rs.most_relevant_pages('10569722', metrics = 'all')

Top 3 (cosine similarity):  ['Travis_Frederick', 'James_Coe', 'Tom_Upton']
Top 3 (euclidean distance):  ['Peter_Somerville', 'John_Lavan', 'Tim_Pawlenty']
Top 3 (pearson correlation):  ['Travis_Frederick', 'James_Coe', 'Tom_Upton']
```

```
In [15]: rs.most_relevant_pages('104239528', metrics = 'all')

Top 3 (cosine similarity):  ['Sabina_Guzzanti', 'Luciano_Ligabue', 'Ivan_Basso']
Top 3 (euclidean distance):  ['Luciano_Ligabue', 'Peter_Diamandis', 'Lovato']
Top 3 (pearson correlation):  ['Sabina_Guzzanti', 'Luciano_Ligabue', 'Ivan_Basso']
```