DM Homework 2

Matteo Emanuele, matricula 1912588

November 2020

1 Exercise 1

In the following exercise we have to implement a search engine using python.
Here we report the way with which we were able to fetch all the elements from the html page of amazon:

```
keyword = 'computer'
X = 6
tokens_from_description = []
final_frames=[]
for i in range(1,X):
    url = 'https://www.amazon.it/s?k=' + keyword +'&page='+ str(i)
    headers = {'User-Agent': 'Chrome/42.0.2311.90 Opera/9.80 (Windows NT 6.0) Presto/2.12.388 Version/12.14'}
    r = requests.get(url, headers = headers)
    #print(r.status_code)
    if r.status_code == 200:
        print('Success!')
    elif r.status_code == 404:
        print('Not Found.')
    elif r.status_code == 503:
        print('too much requests perhaps? Adjust the header!')
    text = r.text
    soup = BeautifulSoup(text, 'html.parser')
    result = soup.findAll(class_='celwidget slot=MAIN template=SEARCH_RESULTS widgetId=search-results')
    temp_df,temp_tokens_from_description = Polish_informations_and_preprocess(result)
    #print(temp_df.shape)
    tokens_from_description += temp_tokens_from_description
    print("len(token_from_descrpition):"+" " + str(len(tokens_from_description)))
    final_frames.append(temp_df)
    time.sleep(5)
if(X>1): df = pd.concat(final_frames)
elif(X==1):
# print(final_frames)
    df = final_frames[0]
#print(df.shape)
del df["Unnamed: 0"]
df.to_csv('amazon_results_polished.csv')
csv_to_tsv("amazon_results_polished.csv","amazon_results_polished.tsv")
```

to briefly explain what it's happening here:
We are iterating for X number of result's pages(this number could have been

higher,but after 6 we will start fetching unrelated results).
We pick the text element from the request we obtained from the http link, and using beautifulSoup we will parse it to html. At this moment,we have an html-parsed text from which we need to fetch the elements.
The products and their respective informations are enclosed in classes and specific div that we fetched in the function "*Polish information and preprocess*" which is reported below:

```
arr_names = []
arr_prices = []
arr_prime = []
arr_stars = []
arr_url = []
arr_tokens = []
for element in result:
    result_names = element.find(class_='a-size-base-plus a-color-base a-text-normal')
    arr_names.append(result_names.text)
    if element.findAll('span', attrs={"class":"a-price-whole"}):
        arr_prices.append(element.find(class_='a-price-whole').text)
        #print(arr_prices)
    if not element.findAll('span', attrs={"class":"a-price-whole"}):
        arr_prices.append("no price available")
    if (len(element.findAll(class_='a-icon a-icon-prime a-icon-medium'))==1 ) :
        arr_prime.append("prime")
    if(len(element.findAll(class_='a-icon a-icon-prime a-icon-medium'))==0):
        arr_prime.append("not prime")
    result_stars = element.findAll(class_='a-row a-size-small')#.findAll(class_='a-row a-size-small')
    if(len(result_stars)!=0):
        arr_stars.append(result_stars[0].find(class_='a-icon-alt').text)
    if(len(result_stars)==0):
        arr_stars.append("no vote available")
    if(element.find('a', {'class': 'a-size-base a-link-normal a-text-normal'}) is None):
        arr_url.append("No url Available")
    if(element.find('a', {'class': 'a-size-base a-link-normal a-text-normal'}) is not None):
        arr_url.append(element.find('a', {'class': 'a-size-base a-link-normal a-text-normal'})['href'])


df,tokenized_filtered_arr_names = tokenize_and_proprocess_data(arr_names,arr_prices,arr_prime,arr_stars,arr_url,only_names)

return df,tokenized_filtered_arr_names
```

Without going into the details, the overall actions that are happening here are fetching elements using specific classes,storing the elements in arrays. The arrays are then used inside another function that will return a dataframe of the elements well-organized togheter with the descriptions already tokenized and preoproccessed.
About the preoprocess we report here the kind of processing we did:

```python
def tokenize_and_proprocess_data(arr_names,arr_prices,arr_prime,arr_stars,arr_url,only_names=False):
    tokenized_filtered_arr_names =[]
    symbol_to_remove = "!@#$%^&*()_-+={}[],'\"”;:.--''xds/|"
    things_to_remove = {"1", "2", "3","4","5","6","7","8","9","0"}
    stop_words = set(stopwords.words('italian'))

    for element in arr_names:
        element = element.lower()
        tokenized_elements = word_tokenize(element,language='italian')
        filtered_sentence = []
        for w in tokenized_elements:
            #if (w not in stop_words and w not in symbol_to_remove):
            if (w not in stop_words and w not in symbol_to_remove and w not in things_to_remove):
                filtered_sentence.append(w)
        tokenized_filtered_arr_names.append(filtered_sentence)
    if(only_names==False):
        # print("print tokenized_filtered_arr_names:")
        # print(len(tokenized_filtered_arr_names))
        helper_list = list(zip(arr_names,arr_prices,arr_prime,arr_stars,arr_url))
        df = pd.DataFrame(helper_list,columns=["description","price","prime?","rating","url link"])
        print("size of df")
        #print(df.shape)
        return df,tokenized_filtered_arr_names
    elif(only_names==True):
        return tokenized_filtered_arr_names
```

I'll discuss briefly the kind of preoprocessing I opted to do in this specific case. We used the stopwords from nltk, but I also built a set of additional elements to be removed. Note that these added elements were discovered studying the description and how they were tokenized(trial-error approach).
At the end of this function,as we said,we return a dataframe of all the elements well organized togheter with the array of tokenized and preprocessed descriptions.
The dataframe that we return will be used at the end to return the elements after the query.
After this we build the inverted index. The inverted index will basically be a dictionary where each key will be a token and assigned to the key there will be tuples composed as follow:

$$\text{tuple} = (\text{document id, tfidf of the word in the document})$$

Here is reported the function to build the inverted index:

```python
def create_inverted_index(hashed_documents):
    dictionary = defaultdict(list)
    idf_dict = dict()
    for i, element in enumerate(hashed_documents):
        max_freq = 0

        for index, token in enumerate(element):
            count = element.count(token)                        # conto quante v
            first_occurence = element.index(token)              # mi salvo la pr
            if count > 1 and first_occurence != index : continue # skippo l'appen

            freq = count / len(element)                         # la frequenza d
            if freq > max_freq : max_freq = freq                # salvo la max f
            dictionary[token].append((i, freq))                 # appendo al diz
            index += 1                                          # index usato pe

        for index, token in enumerate(element):
            count = element.count(token)                        #prendo il count
            first_occurence = element.index(token)
            if count > 1 and first_occurence != index : continue

            list_ = dictionary[token]   #accedo alla tupla di valori assegnati al
            token_index = len(list_) - 1    #
            freq = list_[token_index][1]    #perchè non freq = list_[]
            list_[token_index] = (i, freq / max_freq)   #(id_doc,tf score)

            index += 1

    #qui avro' un dizionario dove ogni key è un token,con assegnate tuple (id_doc
    for key in dictionary:
        count = len(dictionary[key])        #number of doc in which the token app
        idf = math.log2(len(hashed_documents) / count)
        idf_dict[key] = idf

        for i, tuples in enumerate(dictionary[key]): #lista di tuple
            dictionary[key][i] = (tuples[0], tuples[1] * idf)    #tf*idf

    with open("inverted_index.json", "wb") as f:
        f.write(json.dumps(dictionary).encode("utf-8"))

    return dictionary,idf_dict
```

I won't go into the details of the logic of the function. The comments inside the functions will help understanding the logic steps and the solutions adopted for the different problems.
I'll just specify that the tfidf score was computed as the the product of the tf times the idf,where respectively:

$$\text{tf} = \frac{count of the word in the document}{length of the document}$$
$$\text{idf} = \log_2(\frac{number of documents}{number of document in which the word appear})$$

With an inverted index ready,now we can vectorize the documents. We want to bring all the documents to have the same fixed length(number of features) where each feature will be a word in the inverted index, while the assigned value will be the relative tfidf score.

Here is reported the function to vectorize the documents:

```
def vectorize_documents(dictionary, num_documents):
    documents_vectorized = []
    for index in range(num_documents):
        documents_vectorized.append([])

    for index in range(len(documents_vectorized)):
        for key in dictionary:
            doc_found = False
            for tuples in dictionary[key]:
                if tuples[0] == index:
                    doc_found = True
                    documents_vectorized[index].append(tuples[1])
                    break
            if doc_found == False: documents_vectorized[index].append(0)
    return documents_vectorized
```

The final step is approaching. To compute our search algorithm to match the query(computing the cosine similarity) we miss the tokenization and the vectorization of the query. We do this with the following functions:

```
def tokenize_query(query):
    symbol_to_remove = "!@#$%^&*()_-+={}[],'\"”;:.-–''xds/|"
    things_to_remove = {"1", "2", "3","4","5","6","7","8","9","0"}
    stop_words = set(stopwords.words('italian'))
    query = query.lower()
    tokenized_query = word_tokenize(query,language='italian')
    filtered_query = []
    for w in tokenized_query:
        #if (w not in stop_words and w not in symbol_to_remove):
        if (w not in stop_words and w not in symbol_to_remove and w not in things_to_remove):
            filtered_query.append(w)
    return filtered_query
```

```
def vectorize_query(query, dictionary,idf_dict):
    Q_vect = [0] * len(dictionary)

    for q in query:
        c = query.count(q)
        for index,key in enumerate(dictionary):
            if key == q :
                Q_vect[index] = (c/len(query)) *idf_dict[key]
                break
    return Q_vect
```

I want to stress a little bit the fact that the preprocessing part of the query was built to be exactly the same of the other documents. This is important in order to have a ”common ground” between the query and the documents among we are searching.

We are finally ready to compute the cosine similarity:

```
def search_results(query, dictionary, documents_vectorized,idf_dictionary):
    vec_q = vectorize_query(query, dictionary,idf_dictionary)
    # print(" ")
    # print("-----printing the vectorized query:")
    # print(vec_q)
    # print(" ")
    results = compute_cos_sim(vec_q, documents_vectorized)
    # print("-----results of cosine similarity:")
    # print(results)

    out = []
    for i, res in enumerate(results):
        out.append((i, res))
    return list(zip(*sorted(out, key=op.itemgetter(1), reverse=True)))
```

```
def compute_cos_sim(query, documents_vectorized):
    # print(" ")
    # print("----first document vectorized:")
    # print(documents_vectorized[4])
    # print(" ")
    val = []
    for doc_id, document in enumerate(documents_vectorized):
        val.append( np.dot(document, query) / ( np.linalg.norm(query) * np.linalg.norm(document) ) )
    return val
```

Note that in the search results we have already returned a list of tuples conteining the document id(initially in ascending order) togheter with the corresponding cosine similarity score, and at last we sorted with respect to the cosine similarity score. Doing this we can now access the first elements of the tuple to have the id we have to search in the dataframe of the products that we build at the start of the exercise. Here is where we do as such togheter with the query request:

```
Q = "SAMSUNG Galaxy Book Ion NP950XCJ-X01DE Notebook/Portatile Computer Portatile Argento 39,6 cm (15.6"") 19
tokenized_query = tokenize_query(Q)
results, similarities = search_results(tokenized_query, inverted_index, vectorized_documents,idf_dictionary)
print(" ")
print("query requested: " + str(Q))
print(" ")
print("query tokenized: " + str(tokenized_query))
for index,elem in enumerate(results):
    print(df.loc[[elem]])
    print(" ")
    if(index ==10): break # print top 10 ranking
```

Here we report two different queries. The first is a short one, while the second will be a long one:

first query: ”(ricondizionato)”

```
query requested: (ricondizionato)

query tokenized: ['ricondizionato']
                                         description   price    prime?          rating                                         url link
174  PC RICONDIZIONATO HP ELITE 8300 SFF INTEL CORE...  231,00  not prime  4,5 su 5 stelle  /RICONDIZIONATO-HP-ELITE-INTEL-Ricondizionato/...

                                         description   price    prime?          rating                                         url link
155  PC RICONDIZIONATO DELL 7010 SFF Intel Core i5 ...  161,90  not prime  4,5 su 5 stelle  /RICONDIZIONATO-3-20Ghz-LICENZA-Ricondizionato...

                                         description   price prime?          rating                                         url link
220  PC RICONDIZIONATO LENOVO M92P TINY INTEL CORE ...  219,00  prime  4,3 su 5 stelle  /RICONDIZIONATO-LENOVO-INTEL-3470T-Ricondizion...

                                         description   price    prime?          rating                                         url link
218  NOTEBOOK RICONDIZIONATO ULTRASLIM HP FOLIO 947...  429,99  not prime  3,6 su 5 stelle  /NOTEBOOK-RICONDIZIONATO-ULTRASLIM-HP-Ricondiz...

                                        description   price    prime?          rating                                         url link
```

second query: ”SAMSUNG Galaxy Book Ion NP950XCJ-X01DE Notebook/Portatile Computer Portatile Argento 39,6 cm (15.6) 1920 x 1080 Pixel Intel® Core i7 di Decima Generazione 16”

```
                                        description     price prime?          rating                                          url link
73   SAMSUNG Galaxy Book Ion NP950XCJ-X01DE Noteboo...  2.015,89  prime  3,3 su 5 stelle  /SAMSUNG-NP950XCJ-X01DE-Portatile-Generazione-...

                                         description     price prime?          rating                                          url link
199  SAMSUNG Galaxy Book Ion NP930XCJ-K01DE Noteboo...  1.481,61  prime  4,0 su 5 stelle  /SAMSUNG-NP930XCJ-K01DE-Portatile-Generazione-...

                                         description   price prime?          rating                                          url link
169  Acer Aspire 3 A315-56-582U Computer Portatile ...  882,39  prime  no vote available  /Acer-A315-56-582U-Portatile-Generazione-DDR4-...

                                        description               price    prime?            rating         url link
53   ASUS P509JA-EJ025R Grigio Computer portatile 3...  no price available  not prime  no vote available  No url Available

                                        description     price prime?          rating                                          url link
82   Lenovo ThinkBook 15 Grigio Computer Portatile ...  1.084,61  prime  2,7 su 5 stelle  /Lenovo-ThinkBook-Portatile-Generazione-DDR4-S...

                                         description     price prime?          rating                                          url link
188  ASUS ZenBook PRO UX581LV-H2024T Notebook/Porta...  4.057,53  prime  4,0 su 5 stelle  /ASUS-UX581LV-H2024T-Portatile-Generazione-DDR...
```

Note:
The returned results match *perfectly*. Also, the algorithm requires at most 1 - 1.2sec to solve a query.