Introduction to Web Science

Assignment 8

Prof. Dr. Steffen Staab
staab@uni-koblenz.de

René Pickhardt
rpickhardt@uni-koblenz.de

Korok Sengupta
koroksengupta@uni-koblenz.de

Olga Zagovora
zagovora@uni-koblenz.de

Institute of Web Science and Technologies
Department of Computer Science
University of Koblenz-Landau

Submission until: January 11, 2017, 10:00 a.m.
Tutorial on: January 13, 2017, 12:00 p.m.

Please look at all the lessons of part 2 in particular **Similarity of Text** and **graph based models**

For all the assignment questions that require you to write code, make sure to include the code in the answer sheet, along with a separate python file. Where screen shots are required, please add them in the answers directly and not as separate files.

Other than that this sheet is mainly designed to review and apply what you have learnt in part 2 it is a little bit larger but there is also more time over the x-mas break. In any case we wish you a mery x-mas and a happy new year.

Team Name: Golf

Members: Atique Baig, Mtarji Adam, Deepak Garg

1 Similarity - (40 Points)

This assignment will have one exercise which is dived into four subparts. The main idea is to study once again the web crawl of the Simple English Wikipedia. The goal is also to review and aply your knowledge from part 2 of this course.

We have constructed two data sets from it which are all the articles and the link graph extracted from Simple English Wikipedia. The extracted data sets are stored in the file `http://141.26.208.82/store.zip` which contains a pandas container and can be read with pandas in python. In subsection "1.5 Hints" you will find some sample python code that demonstrates how to easily access the data.

With this data set you will create three different models with different similarity measures and finally try to evaluate how similar these models are.

This assignment requires you to handle your data in efficient data structures otherwise you might discover runtime issues. So please read and understand the full assignment sheet with all the tasks that are required before you start implementing some of the tasks.

1.1 Similarity of Text documents (10 Points)

1.1.1 Jaccard - Similarity on sets

1. Build the word sets of each article for each article id.
2. Implement a function `calcJaccardSimilarity(wordset1, wordset2)` that can calculate the jaccard coefficent of two word sets and return the value.
3. Compute the result for the articles `Germany` and `Europe`.

Answer:

```
"""
imports are used on all 4 parts of the assignment
"""
import pandas as pd
import re #used for some text cleaning
from collections import Counter
import math
import time # might need to replace by timeit

store = pd.HDFStore('store2.h5')
df1=store['df1']
df2=store['df2']
```

```
# number of documents or articles
nbr_docs=len(df1.index)
# we create a new column containing the article words
# we apply a function to remove ponctuation and some other characters for
# better word set, we also transform the text to lowercase
# column mainly used to calculate term frequencie
df1['words']=df1.text.apply(lambda x:
    re.sub(r'[,;.\[\]():\*\"\']*','',x).lower().split())
# we also add another column for a set of unique words, column used for jaccard
df1['set']=df1.words.apply(lambda x: set(x))

# function used to calculate the jaccardCoef between 2 wordsets
def calcJaccardSimilarity(wordset1, wordset2):
    # excpetion handling to deal with missing data (inducing division by zero)
    try:
        return len(wordset1.intersection(wordset2))/len(wordset1.union(wordset2))
    except:
        return 0
# the jaccard coef for article 'Germany' and 'Europe'
jaccard_EU_DE=calcJaccardSimilarity(df1[df1.name=="Germany"].iloc[0]['set'],
                                    df1[df1.name=="Europe"].iloc[0]['set'])
print("Jaccard Coef for articles 'Germany' and 'Europe' = %s"%jaccard_EU_DE)
```

1.1.2 TF-IDF with cosine similarity

1. Count the term frequency of each term for each article
2. Count the document frequencies of each term.
3. For each article id provide a dictionary of terms occuring in the article together with their tf-idf scores as the corresponding values.
4. Implement a function `calculateCosineSimilarity(tfIdfDict1, tfIdfDict2)` that computes the cosine similarity for two sparse tf-idf vectors and returns the value.
5. Compute the result for the articles `Germany` and `Europe`.

Answer:

```
# we create a new column to store term frequencies
df1['frequencies']=df1.words.apply(lambda x: Counter(x))
# variable to store document frequencies
doc_frequencies=dict()
for terms in df1.frequencies.tolist():
```

```
    for term in terms:
        if term in doc_frequencies:
            doc_frequencies[term] += 1
        else:
            doc_frequencies[term] = 1

# function used to calculate the tfidf
def calculateTFIDF(tf,df):
    return tf*math.log(nbr_docs/df)

# function used to build a dictionary of tfidf
def buildTFIDFdict(frequencies):
    tfidfDict=dict()
    for term in frequencies:
        tfidfDict[term]=calculateTFIDF(frequencies[term],doc_frequencies[term])
    return tfidfDict

# function used to calculate the euclidian distance based on the tfidf
def docEuclidDist(tfIdfDict):
    return math.sqrt(sum([math.pow(value,2) for key,value in tfIdfDict.items()]))

# adding tfidf column to store the tfidf-dictionaries
df1['tfidf']=df1.frequencies.apply(lambda x: buildTFIDFdict(x))
# we also store the euclidian distance for future use
df1['euclid']=df1.tfidf.apply(lambda x: docEuclidDist(x))

# function used to calculate cosine similarity, we added 2 more parameters
# euclid1 and euclid2 so we can feed our already calculated distances
def calculateCosineSimilarity(tfIdfDict1, tfIdfDict2,euclid1,euclid2):
        # will hold the product of tfidfs
        product_vars=[]
        for term in tfIdfDict1:
            if(term in tfIdfDict2):
                product_vars.append(tfIdfDict1[term]*tfIdfDict2[term])
        sum_=sum(product_vars)
        # non Null vectors
        if(sum_ != 0 and euclid1 != 0 and euclid2 != 0):
            return math.acos(sum_/(euclid1*euclid2))
        else:
            # in case of a null vector we return a pie number
            # will need to think of a better solution
            return 3.14
# calculating cosine similarity between article 'Germany' and 'Europe'
theta=calculateCosineSimilarity(df1[df1.name=="Germany"].iloc[0]['tfidf'],
```

```
                                    df1[df1.name=="Europe"].iloc[0]['tfidf'],
                                    df1[df1.name=="Germany"].iloc[0]['euclid'],
                                    df1[df1.name=="Europe"].iloc[0]['euclid'])
print("Cosine similarity between article 'Germany' and 'Europe' is = %s"%theta)
```

1.2 Similarity of Graphs (10 Points)

You can understand the similarity of two articles by comparing their sets of outlinks (and see how much they have in common). Feel free to reuse the `computeJaccardSimilarity` function from the first part of the exercise. This time do not aply it on the set of words within two articles but rather on the set of outlinks being used within two articles. Again compute the result for the articles `Germany` and `Europe`. **Answer:**
Screenshot:

Figure 1: Console output for part 1.1 and 1.2

```
Jaccard Coef for articles 'Germany' and 'Europe' = 0.04566929133858268
Cosine similarity between article 'Germany' and 'Europe' is = 1.4234239178761714
The jaccard graph coef of articles 'Germany' and 'Europe' = 0.27307692307692305
**Applying a query on article 'Germany'
```

```
# applying the jaccardSimilarity function on the outlinks
jaccard_EU_DE_outlinks=calcJaccardSimilarity(set(df2[df2.name=="Germany"].\
                                                 iloc[0]['out_links']),
                                set(df2[df2.name=="Europe"].iloc[0]['out_links']))

print("The jaccard graph coef of articles 'Germany' and 'Europe' = %s"
      %jaccard_EU_DE_outlinks)
```

1.3 How similar have our similarities been? (10 Points)

Having implemented these three models and similarity measures (text with Jaccard, text with cosine, graph with Jaccard) our goal is to understand and quantify what is going on if they are used in the wild. Therefore in this and the next subtask we want to try to give an answere to the following questions.

- Will the most similar articles to a certain article always be the same independent which model we use?
- How similar are these measures to each other? How can you statistically compare them?

Assume you could use the similarity measure to compute the top k most similar articles for each article in the document collection. We want to analyze how different the rankings for these various models are.

Do some research to find a statistical measure (either from the lectures of part 2 or by doing a web search and coming up with something that we haven't discussed yet) that could be used best to compare various rankings for the same object.

Explain in a short text which measure you would use in such an experiment and why you think it is usefull for our task.

Answer:
In order to have an idea of about how will the similarity models behave on a large scale, we test it by querying one article (in this case "Germany") and fetching all articles by their rank using each of the models implemented. We output the 5 most similar article per model/measure for comparison

Figure 2: Console output for the top 5 most similar article to Germany per measure used

```
 Jaccard similarity top 5:
[('Prussia', 0.225), ('World_War_I_9429', 0.21835443037974683), ('Netherlands',
0.2028985507246377), ('Politics_of_Germany_8451', 0.20219435736677116)]

 Cosine similarity top 5:
[('German_reunification', 1.0332190859339578), ('Treaty_of_Versailles_58e2', 1.07460044221158),
('Luftwaffe', 1.1134938133356302), ('Soviet_occupation_zone', 1.1681269864685788)]

 Jaccard graph similarity top 5:
[('Belgium', 0.4008097165991903), ('Lithuania', 0.39215686274509803), ('Bulgaria',
0.38223938223938225), ('Latvia', 0.3798449612403101)]

Computing similarity using 3 methods for 100 article in 3.426218032836914 seconds

Computing 27497 articles could be estimated to last : 942.1071724891664 seconds
```

```python
# function used to rank aticles from most similar
# takes an article as input and 2 dataframes
# calculates 3 measures using the 3 implemented methods
def queryArticles(article,df1_sub,df2_sub):
    # selecting the query article row in both dataframes
    df1_article=df1[df1.name==article].iloc[0]
    df2_article=df2[df2.name==article].iloc[0]
    # will hold jaccard similarity ranked articles
    jsim_ranks=dict()
    # will hold cosine similarity ranked articles
    cosim_ranks=dict()
    # will hold jaccard graph similarity ranked articles
    jsimgraph_ranks=dict()
```

```
    for index, row in df1_sub.iterrows():
        # we skip if its the same article as the one in the query
        if(row['name'] != article):
            # calculating using the jaccard and cosine similarity (dataframe 1)
            jsim_ranks[row['name']]=calcJaccardSimilarity(df1_article['set'],
                                                          row['set'])
            cosim_ranks[row['name']]=calculateCosineSimilarity(
                                            df1_article['tfidf'],
                                                    row['tfidf'],
                                           df1_article['euclid'],
                                                   row['euclid'])

    for index, row in df2_sub.iterrows():
        if(row['name'] != article):
            # calculating using the jaccard graph similarity (dataframe 2)
            jsimgraph_ranks[row['name']]=calcJaccardSimilarity(
                        set(df2_article['out_links']),set(row['out_links']))

    # sorting using the rank value
    sorted_jsim_ranks=sorted(jsim_ranks.items(),
                                     key=lambda x: x[1],reverse=True)
    sorted_cosim_ranks=sorted(cosim_ranks.items(),
                                      key=lambda x: x[1])
    sorted_jsimgraph_ranks=sorted(jsimgraph_ranks.items(),
                                          key=lambda x: x[1],reverse=True)

    return sorted_jsim_ranks,sorted_cosim_ranks,sorted_jsimgraph_ranks

# calling the query function on article "Germany"
sorted_jsim_ranks,sorted_cosim_ranks,\
                        sorted_jsimgraph_ranks=queryArticles("Germany",df1,df2)
# printing the 5 most similar articles given per method
print("**Applying a query on article 'Germany'\n")
print("*Fetching top 5 most similar articles per measure\n")
print(" Jaccard similarity top 5: \n%s\n"%sorted_jsim_ranks[0:4])
print(" Cosine similarity top 5: \n%s\n"%sorted_cosim_ranks[0:4])
print(" Jaccard graph similarity top 5: \n%s\n"%sorted_jsimgraph_ranks[0:4])
```

We think that the cosine similarity gives the best similarity judging from the proposed articles, while other models gave similar results, they also had articles that are far from our query.
A statistical measure we found interesting is Kendall rank correlation coefficient (or Kendall's tau coefficient), a coefficient that represents the concordance between two columns of ranked data, it ranges from -1 to 1 (1 been identical) and allows us to do statistical comparison of two similarity measures.
The formula goes as follow:

$$\tau = \frac{(number\quad of\quad concordant\quad pair) - (number\quad of\quad discordant\quad pair)}{(number\quad of\quad concordant\quad pair) + (number\quad of\quad discordant\quad pair)} \tag{1}$$

If the ranks for 2 objects agree, they are called a concordant pair. Our evaluation approach will be to first start by calculating and ranking each article against a 100 articles, then we ll sort the ranked data and compute the Kendall tau for each pair of measures and calculate the average over 100 articles. We believe this approach will go well with our models since it relies mainly on rankings.

1.4 Implement the measure and do the experiment (10 Points)

After you came up with a measure you will most likely run into another problem when you plan to do the experiment.

Since runtime is an issue we cannot compute the similarity for all pairs of articles. Tell us:

1. How many similarity computations would have to be done if you wished to do so?
2. How much time would roughly be consumed to do all of these computations?

A better strategy might be to select a couple of articles for which you could compute your meassure. One strategy would be to select the 100 longest articles. Another strategy might be to randomly select 100 articles from our corpus.

Computer your three similarity measures and evaluate them for these two strategies of selecting test data. Present your results. Will the results depend on the method for selecting articles? What are your findings?

Answer:
We have 27 497 articles, for each article we need to compute 3 measures per article, which means $27497 \times (27496 \times 3) = 2268090048$ computations.
Computing 3 similarity measures for a 100 articles takes 3.4 seconds, we expect the whole computation to take 942 seconds or approximately 15 min .

Figure 3: Console output for the experiment

```
Computing similarity using 3 methods for 100 article in 3.426218032836914 seconds

Computing 27497 articles could be estimated to last : 942.1071724891664 seconds

**Random 100 article strategie results :

 The average Kendall Tau for 'jaccard' vs 'cosine' is 0.3260267069790878
 The average Kendall Tau for 'jaccard' vs 'jaccard graph' is 0.007491946885886284
 The average Kendall Tau for 'cosine' vs 'jaccard graph' is 0.01575849194896814

**Longest 100 article strategie results :

 The average Kendall Tau for 'jaccard' vs 'cosine' is 0.42358194672522465
 The average Kendall Tau for 'jaccard' vs 'jaccard graph' is 0.16763028681816816
 The average Kendall Tau for 'cosine' vs 'jaccard graph' is 0.13292560551434646
```

```python
# we will use the Kendall rank correlation coefficient method to compare
# our 3 similarity measures
# takes as parameter the article, the calculated dataframe and a pair of measures
def calculateKendallTau(article,df_sim,m1,m2):
    # selecting the subject row
    df_row=df_sim[df_sim.name==article].iloc[0]
    # extracting the names from the sorted results of the first measures
    # this is important as it helps with calculation if the first method is
    # ranked properly from highest to lowest rank, and so we use the sorted
    # articles from the first measure to loop on the second measure
    names=[t[0] for t in df_row[m1]]
    # total number of ranks
    ranks=len(names)
    m2_ranks=[]
    # appending the proper rank of each article
    for n in names:
        m2_ranks.append(getRankFromTupleList(df_row[m2],n))

    c=[]# ordered list of number of concordant pairs
    d=[]# ordered list of number of discordant pairs
    # for each rank we check the value with the next ranks
    # if its higher we increment the concordant number
    # if its lower we increment the discordant number
    # we skip counting the last rank as there is no rank below it
    for i in range(0,ranks-1):
        c_cnt=0
        d_cnt=0
```

```
        for j in range(i,ranks-1):
            if m2_ranks[j]>m2_ranks[i]:
                c_cnt+=1
            else:
                d_cnt+=1
        c.append(c_cnt)
        d.append(d_cnt)
    sum_c=sum(c)
    sum_d=sum(d)
    # summing up and calculating tau
    return (sum_c-sum_d)/(sum_c+sum_d)

# calculating the average tau for random 100 strategie
print("**Random 100 article strategie results : \n")
# strategie 1
#100 random articles

df1_random_sample=df1.sample(100)
df2_random_sample=df2[df2.name.isin(df1_random_sample.name)]

#print(df1_sample,df2_sample)

# function used to get an index value from a tuple list
# will be used to get the rank of a specific article
def getRankFromTupleList(l,value):
    for pos,t in enumerate(l):
        if(t[0] == value):
            return pos+1
# creating new dataframe to hold the experiment results
df_ = pd.DataFrame(columns=['name','jaccard','cosim','jgraph'])
rows=[]
ts = time.time()
# applying the query for each article and creating a series for it
for article in df1_random_sample.name:
    jsim,cosim,jgraph=queryArticles(article,df1_random_sample,df2_random_sample)
    rows.append(pd.Series({'name': article,
                           'jaccard': jsim,
                           'cosim': cosim,
                           'jgraph': jgraph}))
ts1=time.time()-ts
print("Computing similarity using 3 methods for 100 article in %s seconds\n"%ts1)
print("Computing %s articles could be estimated to last : %s seconds\n"%(nbr_docs,nbr_doc
# series are appended to the dataframe as rows
df_=df_.append(rows)
```

```
# in case of 'jaccard' vs 'cosine'
ttau_random_jc=0
for a in df1_random_sample.name:
    ttau_random_jc+=calculateKendallTau(a,df_,'jaccard','cosim')
average_tau_random_jc=ttau_random_jc/(len(df1_random_sample.name)-1)
print(" The average Kendall Tau for 'jaccard' vs 'cosine' is %s"
      %average_tau_random_jc)

ttau_random_jjg=0
for a in df1_random_sample.name:
    ttau_random_jjg+=calculateKendallTau(a,df_,'jaccard','jgraph')
average_tau_random_jjg=ttau_random_jjg/(len(df1_random_sample.name)-1)
print(" The average Kendall Tau for 'jaccard' vs 'jaccard graph' is %s"
      %average_tau_random_jjg)

ttau_random_cjg=0
for a in df1_random_sample.name:
    ttau_random_cjg+=calculateKendallTau(a,df_,'cosim','jgraph')
average_tau_random_cjg=ttau_random_cjg/(len(df1_random_sample.name)-1)
print(" The average Kendall Tau for 'cosine' vs 'jaccard graph' is %s\n"
      %average_tau_random_cjg)

# calculating the average tau for longest 100 strategie
print("**Longest 100 article strategie results : \n")
# applying a lenght function on text column and sorting using it
df1=df1.assign(l = df1.text.apply(lambda x : len(x))).sort_values(
                                                        'l',ascending=False)
# we sample the top 100 longest articles
df1_longest_sample=df1[0:99]
df2_longest_sample=df2[df2.name.isin(df1_longest_sample.name)]

# we create a new dataframe for our second experiment
df_2 = pd.DataFrame(columns=['name','jaccard','cosim','jgraph'])
rows_2=[]
for article in df1_longest_sample.name:
    jsim,cosim,jgraph=queryArticles(article,df1_longest_sample,df2_longest_sample)
    rows.append(pd.Series({'name': article,
                           'jaccard': jsim,
                           'cosim': cosim,
                           'jgraph': jgraph}))
df_2=df_2.append(rows)

# in case of 'jaccard' vs 'cosine'
ttau_longest_jc=0
```

```
for a in df1_longest_sample.name:
    ttau_longest_jc+=calculateKendallTau(a,df_2,'jaccard','cosim')
average_tau_longest_jc=ttau_longest_jc/(len(df1_longest_sample.name)-1)
print(" The average Kendall Tau for 'jaccard' vs 'cosine' is %s"
      %average_tau_longest_jc)

ttau_longest_jjg=0
for a in df1_longest_sample.name:
    ttau_longest_jjg+=calculateKendallTau(a,df_2,'jaccard','jgraph')
average_tau_longest_jjg=ttau_longest_jjg/(len(df1_longest_sample.name)-1)
print(" The average Kendall Tau for 'jaccard' vs 'jaccard graph' is %s"
      %average_tau_longest_jjg)

ttau_longest_cjg=0
for a in df1_longest_sample.name:
    ttau_longest_cjg+=calculateKendallTau(a,df_2,'cosim','jgraph')
average_tau_longest_cjg=ttau_longest_cjg/(len(df1_longest_sample.name)-1)
print(" The average Kendall Tau for 'cosine' vs 'jaccard graph' is %s\n"
      %average_tau_longest_cjg)
```

After running the experiment a few times, we notice that regardless of the method, there seems to be a strong correlation between Jaccard and Cosine measures as they display a higher coefficient then the other pairs by a margin.
For the correlation between Jaccard and Jaccard graph, Jaccard graph and Cosine, we notice varying results depending on the method of collection, the coefficient doesn't display a consistently high enough correlation.

1.5 Hints:

1. In oder to access the data in python, you can use the following pice of code:

   ```
   import pandas as pd
   store = pd.HDFStore('store.h5')
   df1=store['df1']
   df2=store['df2']
   ```

2. Variables df1 and df2 are pandas DataFrames which is tabular data structure. df1 consists of article's texts, df2 represents links from Simple English Wikipedia articles. Variables have the following columns:

   - "name" is a name of Simple English Wikipedia article,
   - "text" is a full text of the article "name",
   - "out_links" is a list of article names where the article "name" links to.

3. In general you might want to store the counted results in a file before you do the similarity computations and all the research for the third and fourth subtask. Doing all this counting and preperation might allready take quite some runtime.
4. When computing the sparse tf-idf vectors you might allready want to store the eukleadan length of the vectors. otherwise you might discover runtime issues when computing the length again for each similarity computation.
5. Finding the top similar articles for a given article id requires you to compute the similarity of the given article with comparison to all the other known articles and extract the top 5 similarities. Bare in mind that these are quite a lot of similarity computations! You can expect a runtime to find the top similar articles with respect to one of the methods to be up to 10 seconds. If it takes significant longer then you probably have not used the best data structures handle your data.
6. **Even though many third party libraries exist to do this task with even less computational effort those libraries must not be used.**
7. You can find more information about basic usage of pandas DataFrame in pandas documentation.
8. Here are some usefull examples of operations with DataFrame:

```
import pandas as pd

store = pd.HDFStore('store.h5')#read .h5 file
df1=store['df1']
df2=store['df2']
print df1['name'] # select column "name"
print df1.name # select column "name"
print df1.loc[9] #select row with id equals 9
print df1[5:10] #select rows from 6th to 9th (first row is 0)
print df2.loc[0].out_links #select outlinks of article with id=0

#show all columns where column "name" equals "Germany"
print df2[df2.name=="Germany"]

#show column out_links for rows where name is from list ["Germany","Austria"]
print df2[df2.name.isin(["Germany","Austria"])].out_links

#show all columns where column "text" contains word "good"
print df1[df1.text.str.contains("good")]

#add word "city" to the beginning of each text value
#(IT IS ONLY SHOWS RESULT OF OPERATION, see explanation below!)
print df1.text.apply(lambda x: "city "+x)
```

```
#make all text lower case and split text by spaces
df1[["text"]]=df1.text.str.lower().str.split()

def do_sth(x):
        #here is your function
        #
        #
        return x

#apply do_sth function to text column
#Iit will not change column itself, it will only show the result of aplication
print df1.text.apply(do_sth())

#you always have to assign result to , e.g., column,
#in order it affects your data.
#Some functions indeed can change the DataFrame by
#applying them with argument inplace=True
df1[["text"]]=df1.text.apply(do_sth())

#delete column "text"
df1.drop('text', axis=1, inplace=True)
```

Important Notes

Submission

- Solutions have to be checked into the github repository. Use the directory name `groupname/assignment8/` in your group's repository.
- The name of the group and the names of all participating students must be listed on each submission.
- Solution format: all solutions as *one* PDF document. Programming code has to be submitted as Python code to the github repository. Upload *all* `.py` files of your program! Use `UTF-8` as the file encoding. *Other encodings will not be taken into account!*
- Check that your code compiles without errors.
- Make sure your code is formatted to be easy to read.
  - Make sure you code has consistent indentation.
  - Make sure you comment and document your code adequately in English.
  - Choose consistent and intuitive names for your identifiers.
- Do *not* use any accents, spaces or special characters in your filenames.

Acknowledgment

This latex template was created by Lukas Schmelzeisen for the tutorials of "Web Information Retrieval".

LaTeX

Currently the code can only be build using LuaLaTeX, so make sure you have that installed. If on Overleaf, there's an error, go to settings and change the LaTeXengine to `LuaLaTeX`.