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: Oscar

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`.

```
#This is a special case where im calculating jacaard sim for
#only Germany and Europe. I know it says in the assignmnt to
#build the wordset for all documents, but i dont need it here.
#So i haven't built it. However, I put it in 1.3 because it's
#required.

import pandas as pd
import re
import numpy as np
#Jaccard-Similarity on sets
store = pd.HDFStore('store.h5')
df1=store['df1']
df1['text']=df1['text'].str.lower()
df1.text.replace('', np.nan, inplace=True)
df1.dropna(subset=['text'], inplace=True)
df1.text=df1.text.apply(lambda x: re.findall(r'[0-9a-zA-Z]+', x))
df1.text=df1.text.apply(lambda x: ' '.join(map(str, x)))
```

```

def calcJaccardSimilarity(wordset1, wordset2):
      #length of intersection set
      JK1= len(wordset1 & wordset2)
      #length of union set
      JK2 = len(wordset1 | wordset2)
      #Jaccard Formula
      JK = JK1/float(JK2)
      return JK

#get id-s for articles named Germany and Europe
index1= df1[df1.name=="Germany"].index
index1=index1[0]

index2= df1[df1.name=="Europe"].index
index2=index2[0]

#building sets for each article
set1= set(df1.text[index1].split())
set2=set(df1.text[index2].split())

print(calcJaccardSimilarity(set1,set2),"\n")
```

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`.

```
#This is also a special case solution designed for finding
#cosine similarity between germany and europe where im
#considering my text corpa as only the words involved,
#which are words in the texts of germany and europe article.
#I know it's not theoritcally correct as im supposed to cal-
#culate it for all df1 but it works here because it's too
#many zeros that we don't need. However, in 1.3 I've provided
#a full solution covering all pd1 docs
import pandas as pd
import numpy as np
import re
```

```
from collections import defaultdict
from math import log
from math import sqrt
store = pd.HDFStore('store.h5')
df1=store['df1']
df1['text']=df1['text'].str.lower()
df1.text.replace('', np.nan, inplace=True)
df1.dropna(subset=['text'], inplace=True)
df1.text=df1.text.apply(lambda x: re.findall(r'[0-9a-zA-Z]+', x))
df1.text=df1.text.apply(lambda x: ' '.join(map(str, x)))


def calculateCosineSimilarity(tfIdfDict1, tfIdfDict2):
        tfIdfDict1_mag=0
        tfIdfDict2_mag=0
        vec_mult=0
        for key, value in tfIdfDict1.items():
                value2=tfIdfDict2[key]
                type(value2)
                tfIdfDict1_mag+=(value*value)
                tfIdfDict2_mag+=(value2*value2)

                vec_mult+=(value*value2)
        tfIdfDict1_mag=sqrt(tfIdfDict1_mag)
        tfIdfDict2_mag=sqrt(tfIdfDict2_mag)

        return vec_mult/float(tfIdfDict1_mag*tfIdfDict2_mag)

#get id-s for articles named Germany and Europe
index1= df1[df1.name=="Germany"].index
index1=index1[0]

index2= df1[df1.name=="Europe"].index
index2=index2[0]

tf_idf1 = defaultdict(int)
tf_idf2 = defaultdict(int)

docfrq=defaultdict(int)

#preprocesing both tfidf docs data
for word in df1.text[index1].split():
        docfrq[word] = 0
        tf_idf1[word] =0
        tf_idf2[word] =0


for word in df1.text[index2].split():
        docfrq[word] = 0
        tf_idf1[word] =0
        tf_idf2[word] =0


#calculating tf and idf
for index, row in df1.iterrows():
        flag=False
        l=[]
        for word in row.text.split():
                if(word in docfrq  and word not in l):
                        docfrq[word] +=1
                        l.append(word)

N =len(df1)

for word in df1.text[index1].split():
        tf_idf1[word] +=1

for word in df1.text[index2].split():
        tf_idf2[word] +=1

for key, value in docfrq.items():
        if(tf_idf1[key]!=0):
                tf_idf1[key]=tf_idf1[key]*log(N/float(value))
        if(tf_idf2[key]!=0):
                tf_idf2[key]=tf_idf2[key]*log(N/float(value))

print(calculateCosineSimilarity(tf_idf1,tf_idf2))
```

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`.

```
#same as jacaard in 1.111 but for outlinks

import pandas as pd
import re
import numpy as np

#Jaccard-Similarity on sets
store = pd.HDFStore('store.h5')
df2=store['df2']

```

```
def calcJaccardSimilarity(wordset1, wordset2):
      #length of intersection set
      JK1= len(wordset1 & wordset2)
      #length of union set
      JK2 = len(wordset1 | wordset2)
      #Jaccard Formula
      JK = JK1/float(JK2)
      return JK

#get id-s for articles named Germany and Europe
index1= df2[df2.name=="Germany"].index
index1=index1[0]

index2= df2[df2.name=="Europe"].index
index2=index2[0]

#building sets for each article
set1= set(df2.out_links[index1])
set2=set(df2.out_links[index2])

print(calcJaccardSimilarity(set1,set2),"\n")
```

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:`

1. Concerning the first question whether the most similar articles will be the same or not, the answer as i think is no because the similarity measure can sometime take core different approaches in calculating similarity. For example in the case of jacaard, the method only looks on the two sets at hand without a broader look to the whole corpus. However, cosine similarity takes that in mind with global weights in addition to the two docs being measured. This could leave to small similarities that when piled up can diverse the top ranked results of similarity measures.

2. Concerning question 2, i thought about it and wanted to get a new solution to it. Most 2 important factors in comparison are in my opinion similar docs and their ranking. The solution wasn't so new but i took it from other courses we took in ws. I decided to calculate levenstein distance between ranked results for each similarity approach. The inspiration behind this is looking a the lists of resulting documents as different words of same length and since we want to keep similar docs and their ranking in mind, words levenstein does exactly that. PS: I found that cosine similarity is closer to both jacaard text and graph more than they're closer to each other

```
#This is the most elegant form for all other questions
#in terms of modeling similarity as i do it here for
#all documents given each similarity. In cosine sim
#i followed a different and more realistic approach
#by using only non zero entries in tfidf vectors thus
#you will see that my calculateCosineSimilarity function
#is different from here and for me it's the most generic
#one that can apply on any case
import pandas as pd
import re
import numpy as np
from math import log
from math import sqrt


#Jaccard-Similarity on sets
store = pd.HDFStore('store.h5')
df1=store['df1']
df1['text']=df1['text'].str.lower()
df1.text.replace('', np.nan, inplace=True)
df1.dropna(subset=['text'], inplace=True)
df1.text=df1.text.apply(lambda x: re.findall(r'[0-9a-zA-Z]+', x))
df1.text=df1.text.apply(lambda x: ' '.join(map(str, x)))
df2=store['df2']


#functions

def calcJaccardSimilarity(wordset1, wordset2):
      #length of intersection set
      JK1= len(wordset1 & wordset2)
      #length of union set
      JK2 = len(wordset1 | wordset2)
      #Jaccard Formula
      JK = JK1/float(JK2)
      return JK

def calculateCosineSimilarity(tfIdfDict1, tfIdfDict2):
        tfIdfDict1_mag=tfIdfDict1.values()
        tfIdfDict1_mag=[x**2 for x in tfIdfDict1_mag]
        tfIdfDict1_mag= sqrt(sum(tfIdfDict1_mag))

        tfIdfDict2_mag=tfIdfDict2.values()
        tfIdfDict2_mag=[x**2 for x in tfIdfDict2_mag]
        tfIdfDict2_mag= sqrt(sum(tfIdfDict2_mag))

        vec_mult=0

        set1=set(tfIdfDict1.keys())
        set2=set(tfIdfDict2.keys())
        common_words=list(set1&set2)

        for i in range(len(common_words)):
                vec_mult+=tfIdfDict1[common_words[i]]*tfIdfDict2[common_words[i]]

        return vec_mult/float(tfIdfDict1_mag*tfIdfDict2_mag)

def smart_push(l,item,n):
        if(len(l)<n):
                l.append(item)
        else:
                if(item[0]>l[0][0]):
                        l.remove(l[0])
                        l.append(item)
        l=sorted(l, key=lambda x: x[0])
        return l

def levenshteinDistance(s1, s2):
    if len(s1) > len(s2):
        s1, s2 = s2, s1

    distances = range(len(s1) + 1)
    for i2, c2 in enumerate(s2):
        distances_ = [i2+1]
        for i1, c1 in enumerate(s1):
            if(c1 == c2):
                distances_.append(distances[i1])
            else:
                distances_.append(1 + min((distances[i1], distances[i1 + 1], dista
        distances = distances_
    return distances[-1]


target_doc_index= df1[df1.name=="Germany"].index
target_doc_index=target_doc_index[0]
target_doc_text = df1.text[target_doc_index]
target_doc_outlinks = df2.out_links[target_doc_index]

#cosine similarity pre processing
N=len(df1)

docfrq=dict()
target_doc_tfidf=dict()
docs_tfidf={}

print('started preprocessing')
#filled docfrqs and target_tfidt
for index,row in df1.iterrows():
        docs_tfidf[index]=dict()
        l=[]
        for word in row.text.split():
                if(word in docfrq and word not in l):
                        docfrq[word]+=1
                else:
                        if(word not in docfrq):
                                docfrq[word]=1
                l.append(word)


                if(index==target_doc_index):
                        if(word in target_doc_tfidf):
                                target_doc_tfidf[word]+=1
                        else:
                                target_doc_tfidf[word]=1

                if(word in docs_tfidf[index]):
                        docs_tfidf[index][word]+=1
                else:
                        docs_tfidf[index][word]=1
        if(index%10000==0):
                print(index)

for key,value in target_doc_tfidf.items():
        target_doc_tfidf[key]=value*log(N/float(docfrq[key]))


text_jacard=[]
graph_jacard=[]
```

```
cosine=[]
print('preprocessing ended ... calculating similarity')
for index,row in df1.iterrows():
        #calculating cosine
        for key,value in docs_tfidf[index].items():
                docs_tfidf[index][key]=value*log(N/float(docfrq[key]))
        cosine=smart_push(cosine,(calculateCosineSimilarity(target_doc_tfidf,docs_

        #calculating text jacaard
        target_text_set=set(target_doc_text.split())
        current_text_set=set(df1.text[index].split())
        text_jacard= smart_push(text_jacard,(calcJaccardSimilarity(target_text_set
        #calculating graph jacaard
        target_graph_set=set(target_doc_outlinks)
        current_graph_set=set(df2.out_links[index])
        graph_jacard=smart_push(graph_jacard,(calcJaccardSimilarity(target_graph_s
        if(index %10000==0):
                print(index)

#getting the docs ids ranked according to each sim measure
text_jacard = [int(i[1]) for i in text_jacard]
cosine = [int(i[1]) for i in cosine]
graph_jacard = [int(i[1]) for i in graph_jacard]

print(text_jacard)
print(cosine)
print(graph_jacard)

#grouping the output to calculate levenstein distance between each measure
#by using an function for levenstein distance i found online. I assign
#each doc with a letter and this way each ranked result will be like a word
#and at the end we calculate levenstein distance for all

grouped=list(set(text_jacard)|set(cosine)|set(graph_jacard))
reference=dict()
for i in range(len(grouped)):
        reference[grouped[i]]=chr(97+i)

text_jacard_string=''
cosine_string=''
graph_jacard_string=''
for i in range(len(text_jacard)):
        text_jacard_string+=reference[text_jacard[i]]
        cosine_string+=reference[cosine[i]]
        graph_jacard_string+=reference[graph_jacard[i]]


print(str(levenshteinDistance(text_jacard_string,cosine_string))+' levenstein dis
print(str(levenshteinDistance(text_jacard_string,graph_jacard))+' levenstein dista
print(str(levenshteinDistance(cosine_string,graph_jacard_string))+' levenstein di
```

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? `Answer:` 1)Calculating cosine sim o(n) n=nmbr of docs (chosen docs) 2)Saving the result and sorting them o(n)log(n) considering that the sorter i used in python is efficient or else it would be o(n pow 2)
2. How much time would roughly be consumed to do all of these computations? `Answer:` our complexity = o(n) + o(n)log(n) our time= o(n)time + o(n)log(n)time assuming out time is 1 second to execute anything which is far not accurate for sure. PS: n is roughly 27000 so the time = 27000 + 27000*4.43=5.43*27000=146610 seconds=2443.5 minutes=40.725 hours=1.7 days

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:`I found out that selecting the longest 100 docs is more informative that just randomly selecting docs, especially in cosine simulation. Longer articles bigger chances for similarity values accumulation. It's rather more simulating to reality than random docs.

PS:I made two files, one for random 100 and one for longest articles.

Random

```
#Solution for 100 randomly selected docs
#This solution is also a special case solution
#where im considering my docs corpa to be the
#selected items + germany article

import pandas as pd
import re
import numpy as np
import copy
from math import log
from math import sqrt

#Jaccard-Similarity on sets
store = pd.HDFStore('store.h5')
df1=store['df1']
df1['text']=df1['text'].str.lower()
df1.text.replace('', np.nan, inplace=True)
df1.dropna(subset=['text'], inplace=True)
df1.text=df1.text.apply(lambda x: re.findall(r'[0-9a-zA-Z]+', x))
df1.text=df1.text.apply(lambda x: ' '.join(map(str, x)))
df2=store['df2']


#functions

def calcJaccardSimilarity(wordset1, wordset2):
      #length of intersection set
      JK1= len(wordset1 & wordset2)
      #length of union set
      JK2 = len(wordset1 | wordset2)
      #Jaccard Formula
      JK = JK1/float(JK2)
      return JK

def calculateCosineSimilarity(tfIdfDict1, tfIdfDict2):
        tfIdfDict1_mag=0
        tfIdfDict2_mag=0
        vec_mult=0
        for key, value in tfIdfDict1.items():
                value2=tfIdfDict2[key]
                type(value2)
                tfIdfDict1_mag+=(value*value)
                tfIdfDict2_mag+=(value2*value2)

                vec_mult+=(value*value2)
        tfIdfDict1_mag=sqrt(tfIdfDict1_mag)
        tfIdfDict2_mag=sqrt(tfIdfDict2_mag)
        return vec_mult/float(tfIdfDict1_mag*tfIdfDict2_mag)

def levenshteinDistance(s1, s2):
    if len(s1) > len(s2):
        s1, s2 = s2, s1

    distances = range(len(s1) + 1)
    for i2, c2 in enumerate(s2):
        distances_ = [i2+1]
        for i1, c1 in enumerate(s1):
            if(c1 == c2):
                distances_.append(distances[i1])
            else:
                distances_.append(1 + min((distances[i1], distances[i1 + 1], dista
        distances = distances_
    return distances[-1]
random_samples = df1.sample(n=100)
target_doc_index= df1[df1.name=="Germany"].index
target_doc_index=target_doc_index[0]
target_doc_text = df1.text[target_doc_index]
target_doc_outlinks = df2.out_links[target_doc_index]

#cosine similarity pre processing
N=len(df1)
docfrq=dict()
target_doc_tfidf=dict()
docs_tfidf={}
for index,row in random_samples.iterrows():
        docs_tfidf[index]={}
        for word in row.text.split():
                docfrq[word]=0
                target_doc_tfidf[word]=0
                docs_tfidf[index][word]=0

for word in target_doc_text.split():
        if(word in docfrq):
                docfrq[word]+=1
        else:
                docfrq[word]=1
        if(word in target_doc_tfidf):
                target_doc_tfidf[word]+=1
        else:
                target_doc_tfidf[word]=1

for index,row in random_samples.iterrows():
        docs_tfidf[index]=copy.copy(docfrq)

for index,row in random_samples.iterrows():
        for word in row.text.split():
                if(index==target_doc_index):
                        target_doc_tfidf[word]+=1
                else:
                        docs_tfidf[index][word]+=1


for index, row in df1.iterrows():
        flag=False
        l=[]
        for word in row.text.split():
                if(word in docfrq  and word not in l):
                        docfrq[word] +=1
                        l.append(word)

print("Data preprocessing is over")

text_jacard=[]
graph_jacard=[]
cosine=[]


for index, row in random_samples.iterrows():
        #calculating text jacaard
        target_text_set=set(target_doc_text.split())
        current_text_set=set(df1.text[index].split())
        text_jacard.append((calcJaccardSimilarity(target_text_set,current_text_se

        #calculating graph jacaard
        target_graph_set=set(target_doc_outlinks)
        current_graph_set=set(df2.out_links[index])
        graph_jacard.append((calcJaccardSimilarity(target_graph_set,current_graph_

        #calculating cosine sim
        tf_idf1=target_doc_tfidf
        tf_idf2=docs_tfidf[index]
        for key, value in docfrq.items():
                if(tf_idf1[key]!=0):
                        tf_idf1[key]=tf_idf1[key]*log(N/float(value))
                if(tf_idf2[key]!=0):
                        tf_idf2[key]=tf_idf2[key]*log(N/float(value))
        cosine.append((calculateCosineSimilarity(tf_idf1,tf_idf2),index))


text_jacard.sort(key=lambda tup: tup[0])
graph_jacard.sort(key=lambda tup: tup[0])
cosine.sort(key=lambda tup: tup[0])

#getting the docs ids ranked according to each sim measure
text_jacard = [int(i[1]) for i in text_jacard]
cosine = [int(i[1]) for i in cosine]
graph_jacard = [int(i[1]) for i in graph_jacard]

print(text_jacard)
print(graph_jacard)
print(cosine)


#grouping the output to calculate levenstein distance between each measure
#by using an function for levenstein distance i found online. I assign
#each doc with a letter and this way each ranked result will be like a word
#and at the end we calculate levenstein distance for all
```

```

grouped=list(set(text_jacard)|set(cosine)|set(graph_jacard))
reference=dict()
for i in range(len(grouped)):
        reference[grouped[i]]=chr(97+i)

text_jacard_string=''
cosine_string=''
graph_jacard_string=''
for i in range(len(text_jacard)):
        text_jacard_string+=reference[text_jacard[i]]
        cosine_string+=reference[cosine[i]]
        graph_jacard_string+=reference[graph_jacard[i]]


print(str(levenshteinDistance(text_jacard_string,cosine_string))+' levenstein dis
print(str(levenshteinDistance(text_jacard_string,graph_jacard))+' levenstein dista
print(str(levenshteinDistance(cosine_string,graph_jacard_string))+' levenstein di
```

Longest

```
#Solution for longest 100 articles
#This solution is also a special case solution
#where im considering my docs corpa to be the
#selected items + germany article

import pandas as pd
import re
import numpy as np
import copy
from math import log
from math import sqrt

#Jaccard-Similarity on sets
store = pd.HDFStore('store.h5')
df1=store['df1']
df1['text']=df1['text'].str.lower()
df1.text.replace('', np.nan, inplace=True)
df1.dropna(subset=['text'], inplace=True)
df1.text=df1.text.apply(lambda x: re.findall(r'[0-9a-zA-Z]+', x))
df1.text=df1.text.apply(lambda x: ' '.join(map(str, x)))
df2=store['df2']


#functions

def calcJaccardSimilarity(wordset1, wordset2):
      #length of intersection set
      JK1= len(wordset1 & wordset2)
      #length of union set
      JK2 = len(wordset1 | wordset2)
      #Jaccard Formula
      JK = JK1/float(JK2)
      return JK

def calculateCosineSimilarity(tfIdfDict1, tfIdfDict2):
        tfIdfDict1_mag=0
        tfIdfDict2_mag=0
        vec_mult=0
        for key, value in tfIdfDict1.items():
                value2=tfIdfDict2[key]
                type(value2)
                tfIdfDict1_mag+=(value*value)
                tfIdfDict2_mag+=(value2*value2)

                vec_mult+=(value*value2)
        tfIdfDict1_mag=sqrt(tfIdfDict1_mag)
        tfIdfDict2_mag=sqrt(tfIdfDict2_mag)
        return vec_mult/float(tfIdfDict1_mag*tfIdfDict2_mag)

def levenshteinDistance(s1, s2):
    if len(s1) > len(s2):
        s1, s2 = s2, s1

    distances = range(len(s1) + 1)
    for i2, c2 in enumerate(s2):
        distances_ = [i2+1]
        for i1, c1 in enumerate(s1):
            if(c1 == c2):
                distances_.append(distances[i1])
            else:
                distances_.append(1 + min((distances[i1], distances[i1 + 1], dista
        distances = distances_
    return distances[-1]

#I add a length column to the df1 to sort it
lengths=[]
for index,row in df1.iterrows():
        lengths.append(len(df1.text[index]))
df1['length']=lengths
df1=df1.sort_values(by=['length'], ascending=[False])
samples=df1.head(100)


target_doc_index= df1[df1.name=="Germany"].index
target_doc_index=target_doc_index[0]
target_doc_text = df1.text[target_doc_index]
target_doc_outlinks = df2.out_links[target_doc_index]

```

```
#cosine similarity pre processing
N=len(df1)
docfrq=dict()
target_doc_tfidf=dict()
docs_tfidf={}
for index,row in samples.iterrows():
        docs_tfidf[index]={}
        for word in row.text.split():
                docfrq[word]=0
                target_doc_tfidf[word]=0
                docs_tfidf[index][word]=0

for word in target_doc_text.split():
        if(word in docfrq):
                docfrq[word]+=1
        else:
                docfrq[word]=1
        if(word in target_doc_tfidf):
                target_doc_tfidf[word]+=1
        else:
                target_doc_tfidf[word]=1

for index,row in samples.iterrows():
        docs_tfidf[index]=copy.copy(docfrq)

for index,row in samples.iterrows():
        for word in row.text.split():
                if(index==target_doc_index):
                        target_doc_tfidf[word]+=1
                else:
                        docs_tfidf[index][word]+=1


for index, row in df1.iterrows():
        flag=False
        l=[]
        for word in row.text.split():
                if(word in docfrq  and word not in l):
                        docfrq[word] +=1
                        l.append(word)

print("Data preprocessing is over")

text_jacard=[]
graph_jacard=[]
cosine=[]


for index, row in samples.iterrows():
        #calculating text jacaard
        target_text_set=set(target_doc_text.split())
        current_text_set=set(df1.text[index].split())
        text_jacard.append((calcJaccardSimilarity(target_text_set,current_text_set

        #calculating graph jacaard
        target_graph_set=set(target_doc_outlinks)
        current_graph_set=set(df2.out_links[index])
        graph_jacard.append((calcJaccardSimilarity(target_graph_set,current_graph_

        #calculating cosine sim
        tf_idf1=target_doc_tfidf
        tf_idf2=docs_tfidf[index]
        for key, value in docfrq.items():
                if(tf_idf1[key]!=0):
                        tf_idf1[key]=tf_idf1[key]*log(N/float(value))
                if(tf_idf2[key]!=0):
                        tf_idf2[key]=tf_idf2[key]*log(N/float(value))
        cosine.append((calculateCosineSimilarity(tf_idf1,tf_idf2),index))


text_jacard.sort(key=lambda tup: tup[0])
graph_jacard.sort(key=lambda tup: tup[0])
cosine.sort(key=lambda tup: tup[0])

#getting the docs ids ranked according to each sim measure
text_jacard = [int(i[1]) for i in text_jacard]
cosine = [int(i[1]) for i in cosine]
graph_jacard = [int(i[1]) for i in graph_jacard]

print(text_jacard)
print(graph_jacard)
print(cosine)


#grouping the output to calculate levenstein distance between each measure
#by using an function for levenstein distance i found online. I assign
#each doc with a letter and this way each ranked result will be like a word
#and at the end we calculate levenstein distance for all

grouped=list(set(text_jacard)|set(cosine)|set(graph_jacard))
reference=dict()
for i in range(len(grouped)):
        reference[grouped[i]]=chr(97+i)

text_jacard_string=''
cosine_string=''
graph_jacard_string=''
for i in range(len(text_jacard)):
        text_jacard_string+=reference[text_jacard[i]]
        cosine_string+=reference[cosine[i]]
        graph_jacard_string+=reference[graph_jacard[i]]


print(str(levenshteinDistance(text_jacard_string,cosine_string))+' levenstein dis
print(str(levenshteinDistance(text_jacard_string,graph_jacard))+' levenstein dista
print(str(levenshteinDistance(cosine_string,graph_jacard_string))+' levenstein di
```

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`.