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: Bravo

Shriharsh Ambhore Kandhasamy Rajasekaran Daniel Akbari

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`.

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`.

```
1:
2:
3: # coding: utf-8
4:
5: import pandas as pd
6: import numpy as np
7: import sys
8: from collections import  Counter
9: import math
10:
11:
12: store=pd.HDFStore("store2.h5")
13: df1=store['df1']
14:
15: #split on space for each and every article
16:
17: # following code creates a word set for each and every article
18: copyDF=df1.copy()
19: copyDF['wordset']=copyDF.text.map(lambda x: set(x.lower().split()))
20:
21:
22: def calcJaccardSimilarity(wordset1, wordset2):
23:     wordset1=set(wordset1)
24:     wordset2=set(wordset2)
25:     inter=wordset1.intersection(wordset2)
26:     union=wordset1.union(wordset2)
27:     jc=(len(inter)/len(union))
28:     return jc
29:
30: Germany=copyDF['wordset'][copyDF['name']=='Germany']
31: Europe=copyDF['wordset'][copyDF['name']=='Europe']
32:
33: #print(Germany.tolist()[0])
34: #print('*****words in article splitted on space*****')
35: print('Jaccard coefficent for articles Germany and Europe:',calcJaccardSimilarity
36:
37:
38: # In order to find cosine similarity - we need term frequency, document frequency
39: # tf-idf score
40: # following creates term frequency
41: copyDF['termfrequency']=copyDF.text.map(lambda x: Counter(x.lower().split()).items
42:
43:
44: # it goes through each article's wordset and find the document freq
45: def temp_docfreq_function(rows):
46:     dictofwords = {}
47:
48:     for row in rows:
49:         #print(row)
```

```
50:         for word in row:
51:             if word in dictofwords:
52:                 num=dictofwords[word]+1
53:             else:
54:                 num=1
55:             dictofwords[word]=num
56:
57:     return dictofwords
58:
59: docFreqDict=temp_docfreq_function(copyDF['wordset'])
60:
61: d=len(copyDF['name']) # row size
62:
63: def temp_tfidf_function(wordHist):
64:     ##tf-idf= tf of word * (number of documents/df(word))
65:     numofdocs=d
66:     dictofwords = {}
67:     for word in wordHist:
68:         #word[0] - is the word (key)
69:         #word[1] - is the term freq value
70:         if word[0] in docFreqDict:
71:             #print(word)
72:
73:             df=docFreqDict[word[0]] # get the document frequency of that word
74:             idf=math.log((numofdocs/df),10) # calculate the idf
75:             tfidf=idf*word[1]  # calculate the tf-idf= tf(word)*idf(word)
76:             dictofwords[word[0]]=tfidf
77:
78:     return dictofwords
79:
80: #tfidfDict=temp_tfidf_function(copyDF['termfrequency'][0],len(copyDF['name']))
81: ## creating a coulum tfidf for each word in an article
82: copyDF['tfidf']=copyDF.termfrequency.map(temp_tfidf_function)
83:
84: def calculateCosineSimilarity(doc1, doc2):
85:
86:     temp=doc1.iteritems()
87:     temp2=doc2.iteritems()
88:     dict1=None
89:     dict2=None
90:     # the data structure is a list with two elements and second element is the di
91:     # that we wanted
92:     for t2 in temp2:
93:         dict2=t2[1]
94:     for t in temp:
95:         dict1=(t[1])
96:     dotprod=0
97:     for k1 in dict1:
98:         if k1 in dict2.keys():
```

```
99:             dotprod=dotprod+(dict1[k1]*dict2[k1])
100:
101:     dict1dot=0
102:     dict2dot=0
103:
104:     for key1 in dict1:
105:         dict1dot=dict1dot+(dict1[key1]*dict1[key1])
106:
107:     for key2 in dict2:
108:         dict2dot=dict2dot+(dict2[key2]*dict2[key2])
109:
110:     cosinesimilarity=dotprod/((math.sqrt(dict1dot))*(math.sqrt(dict2dot)))
111:
112:     return cosinesimilarity
113:
114:
115:
116: CSGermany=copyDF['tfidf'][copyDF['name']=='Germany']
117: CSEurope=copyDF['tfidf'][copyDF['name']=='Europe']
118:
119: #print(CSEurope)
120:
121:
122:
123: print('Cosine Similarity of articles Germany and Europe:',calculateCosineSimilarit
124: print(type(CSGermany))
125: print(type(CSEurope))
```

```
1: >>> runfile('/home/kandy/koblenz-web-science/winter-2016/intro-web-science/bravo/a
2: Jaccard coefficent for articles Germany and Europe: 0.03504043126684636
3: Cosine Similarity of articles Germany and Europe: 0.09776700656094285
```

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`.

```
1:
2: # coding: utf-8
3:
4: import pandas as pd
5: from bravo_assignment8_q1_1 import calcJaccardSimilarity
6:
7: store=pd.HDFStore("store2.h5")
```

```
#df1=store['df1']
out_link_df=store['df2']
#print(out_link_df.head)
print(out_link_df.columns)


Germany =out_link_df['out_links'][out_link_df['name']=='Germany']
Europe = out_link_df['out_links'][out_link_df['name']=='Europe']

germany_list = Germany.tolist()[0]
europe_list = Europe.tolist()[0]

print('Based on Outlinks : Jaccard coefficent for articles Germany and Europe :',
      calcJaccardSimilarity(germany_list, europe_list))
```

```
>>> runfile('/home/kandy/koblenz-web-science/winter-2016/intro-web-science/bravo/a

Based on Outlinks : Jaccard coefficent for articles Germany and Europe : 0.273076
```

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) Depending on the similarity measure used, the most similar article to a certain article will vary. There could be some overlaps based on the technique being used. If we consider

the cosine similarity and Jaccard similarity using text, there could be same values for certain article because of similarity of techniques being used to certain degree (we consider the intersection of words in both cases - in cosine, extra information, which is word occurrence is also considered). But consider the technique Jaccard similarity based on out links, it operates out entirely different from the other techniques discussed above. If at all, we get the same article as most similar then it is purely out of chance.

We did find that the most similar article for 'German' article is different between Jaccard similarity based on out links and others.

2) Kendall's Tau method is a good measure to compare the ordinal rankings between different methodologies. The score ranges from -1 to 1. 1 - specifying a strong ranking similarity -1 - specifying a string anti-ranking similarity 0 - being independent

It is found out by ratio of subtraction of concordant pairs and non-concordant pairs to sum of concordant and non-cordant pairs. (C-D)/(C+D)

Let us say two similarity measures were used for 'German' article against all other articles. The measurements would be different for each and every article. Sort one of those similarity measures in descending with highest rank in top and keep the rest as it is. Take each row and compare if there is increase or decrease in rank in the same similarity then does it accompany by an increase or decrease in rank for the other similartiy technique. If so then consider it is a concordant pair otherwise discordant. Count the no. of concordant pairs and discordant pairs and apply the above formula

A similar approach would be to use kendalls Tau B which considers the case of equal ranking and discards it from the formula.

In the program, we have computed similarity measures (cosine using text and jaccard using text and out links) for 'German' article against first 75 articles. we compared these similarity measures using kendalls tau and computed the score. From our results, jaccard using text and out links seem to be having closer ranking similarity.

```
 1:
 2: # coding: utf-8
 3:
 4: import pandas as pd
 5: import numpy as np
 6: import sys
 7: from collections import  Counter
 8: import math
 9:
10: #import bravo_assignment8_q1_1 as q1
11:
12: store=pd.HDFStore("store2.h5")
13: text_df = store['df1']
14: out_link_df=store['df2']
15:
16: corpus_size = 75
```

```
17:
18: text_df = text_df.iloc[0:corpus_size]
19: #print(text_df.head)
20: out_link_df = out_link_df.iloc[0:corpus_size]
21:
22:
23: # following code creates a word set for each and every article
24: text_df['wordset']=text_df.text.map(lambda x: set(x.lower().split()))
25:
26:
27: def calcJaccardSimilarity(wordset1, wordset2):
28:     wordset1=set(wordset1)
29:     wordset2=set(wordset2)
30:     inter=wordset1.intersection(wordset2)
31:     union=wordset1.union(wordset2)
32:     jc=(len(inter)/len(union))
33:     return jc
34:
35:
36: text_df['termfrequency']=text_df.text.map(lambda x: Counter(x.lower().split()).ite
37:
38: # it goes through each article's wordset and find the document freq
39: def temp_docfreq_function(rows):
40:     dictofwords = {}
41:
42:     for row in rows:
43:         #print(row)
44:         for word in row:
45:             if word in dictofwords:
46:                 num=dictofwords[word]+1
47:             else:
48:                 num=1
49:             dictofwords[word]=num
50:
51:     return dictofwords
52:
53: docFreqDict=temp_docfreq_function(text_df['wordset'])
54:
55: d=len(text_df['name']) # row size
56:
57: def temp_tfidf_function(wordHist):
58:     ##tf-idf= tf of word * (number of documents/df(word))
59:     numofdocs=d
60:     dictofwords = {}
61:     for word in wordHist:
62:         #word[0] - is the word (key)
63:         #word[1] - is the term freq value
64:         if word[0] in docFreqDict:
65:             #print(word)
```

```
66:
67:             df=docFreqDict[word[0]] # get the document frequency of that word
68:             idf=math.log((numofdocs/df),10) # calculate the idf
69:             tfidf=idf*word[1]  # calculate the tf-idf= tf(word)*idf(word)
70:             dictofwords[word[0]]=tfidf
71:
72:     return dictofwords
73:
74: ## creating a coulum tfidf for each word in an article
75: text_df['tfidf']=text_df.termfrequency.map(temp_tfidf_function)
76:
77: def calculateCosineSimilarity(dict1, dict2):
78:     dotprod=0
79:     for k1 in dict1:
80:         if k1 in dict2.keys():
81:             dotprod=dotprod+(dict1[k1]*dict2[k1])
82:
83:     dict1dot=0
84:     dict2dot=0
85:
86:     for key1 in dict1:
87:         dict1dot=dict1dot+(dict1[key1]*dict1[key1])
88:
89:     for key2 in dict2:
90:         dict2dot=dict2dot+(dict2[key2]*dict2[key2])
91:
92:     if(dict1dot == 0 or dict2dot == 0):
93:         cosinesimilarity = 1
94:     else :
95:         cosinesimilarity=dotprod/((math.sqrt(dict1dot))*(math.sqrt(dict2dot)))
96:
97:     return (1 - cosinesimilarity) # we wanted to reverse it to compare it with ja
98:
99: #print('-------------------------------------------------------------------------')
100:
101: #print(text_df['name'])
102: #print(out_link_df['name'])
103:
104: germanJackardListRank = []
105: germanCosineSimilarityListRank = []
106: germanJackardListRank_links = []
107: germanRow = text_df[text_df['name']=='German']
108: germanRow_link = out_link_df[out_link_df['name']=='German']
109: j = 0
110: rows_count = text_df.shape[0]
111:
112: while (j < rows_count):
113:     row = text_df.iloc[j]
114:     out_link_row = out_link_df.iloc[j]
```

```
115:     germanCosineSimilarityListRank.append(calculateCosineSimilarity(row['tfidf'],
116:     germanJackardListRank.append(calcJaccardSimilarity(row['wordset'], germanRow[
117:     germanJackardListRank_links.append(calcJaccardSimilarity(out_link_row['out_lin
118:     j = j + 1
119:
120:
121:
122: #print(germanCosineSimilarityListRank)
123: #print(germanJackardListRank)
124:
125: def kendalls_tau(list1, list2):
126:     score = 0
127:     con_pair = 0
128:     non_con_pair = 0
129:
130:     length = len(list1)
131:
132:     if(length > len(list2)) :
133:         length = len(list2)
134:
135:     i = 0
136:     while(i < length) :
137:         j = i + 1
138:         while(j < length) :
139:             if(((list1[i] - list1[j]) * (list2[i] - list2[j])) >= 0):
140:                 con_pair = con_pair + 1
141:             else:
142:                 non_con_pair = non_con_pair + 1
143:             j = j + 1
144:         i += 1
145:
146:     score = (con_pair - non_con_pair) / (con_pair + non_con_pair)
147:     return score
148:
149: print('kendalls Tau score for text based cosine and jaccard similarity - ', kendal
150: print('kendalls Tau score for text based cosine and link based jaccard similarity
151: print('kendalls Tau score for text based jaccard and link based jaccard similarity
152:
153: '''
154: for row in text_iterate:
155:     #print(row[0])
156:     #print(row[1].tfidf)
157:     print(type(germanRow['tfidf']))
158:     print(type(row[1]['tfidf']))
159:     germanListRank.append(calculateCosineSimilarity(germanRow['tfidf'], row[1]['t
160:     #print(type(row))
161:     break
162: '''
```

```
1: >>> runfile('/home/kandy/koblenz-web-science/winter-2016/intro-web-science/bravo/a
2: Reloaded modules: bravo_assignment8_q1_1
3: kendalls Tau score for text based cosine and jaccard similarity -
   -0.07603603603603604
4: kendalls Tau score for text based cosine and link based jaccard similarity -
   0.11135135135135135
5: kendalls Tau score for text based jaccard and link based jaccard similarity -
   0.2627027027027027
```

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:
1) Let us say that we have 'n' articles. If we want to compare every article with other then we can say that n * (n-1) comparisons need to be done. But, if we consider the properties of similarity measures
* comparison with the same article results with value 1
* comparison of a with b is same as comparison of b with a
By considering the above two instances, it would take about n* (n-1)/2
number of articles - 27497. 378028756 comparisons need to be done
2) It takes about 1.6 milli seconds to run one cosine similarity comparison. So if we have to do as many comparisons as stated above then it will take about 7 days to do it. Combining other similarities and involving kendalls tau ranking similarity measures will only increase the time complexity

We did the 3 similartiy measures based on random sampling and longest articles both 100s in number. We found that the results do vary depending upon the method of selecting articles.

```
 1:
 2: # coding: utf-8
 3:
 4: import pandas as pd
 5: import numpy as np
 6: import sys
 7: from collections import  Counter
 8: import math
 9: import time
10:
11: #import bravo_assignment8_q1_1 as q1
12:
13: store=pd.HDFStore("store2.h5")
14: #print(text_df.shape)
15: corpus_size = 100
16: text_df = store['df1']
17: out_link_df=store['df2']
18:
19:
20: #text_df = text_df.iloc[0:corpus_size]
21: #print(text_df.head)
22: #out_link_df = out_link_df.iloc[0:corpus_size]
23:
24: ## random 100 and 100long articles
25: # following code creates a word set for each and every article
26: text_df['wordset']=text_df.text.map(lambda x: set(x.lower().split()))
27:
28:
29:
30:
31:
32:
33: #out_link_df_longest_article;
34:
35:
36:
37: def calcJaccardSimilarity(wordset1, wordset2):
38:     wordset1=set(wordset1)
39:     wordset2=set(wordset2)
40:     inter=wordset1.intersection(wordset2)
41:     union=wordset1.union(wordset2)
42:     jc=(len(inter)/len(union))
43:     return jc
44:
45:
46: text_df['termfrequency']=text_df.text.map(lambda x: Counter(x.lower().split()).ite
47:
48:
49:
```

```
50:
51:
52:
53: # it goes through each article's wordset and find the document freq
54: def temp_docfreq_function(rows):
55:     dictofwords = {}
56:
57:     for row in rows:
58:         #print(row)
59:         for word in row:
60:             if word in dictofwords:
61:                 num=dictofwords[word]+1
62:             else:
63:                 num=1
64:             dictofwords[word]=num
65:
66:     return dictofwords
67:
68: docFreqDict=temp_docfreq_function(text_df['wordset'])
69:
70: d=len(text_df['name']) # row size
71:
72: def temp_tfidf_function(wordHist):
73:     ##tf-idf= tf of word * (number of documents/df(word))
74:     numofdocs=d
75:     dictofwords = {}
76:     for word in wordHist:
77:         #word[0] - is the word (key)
78:         #word[1] - is the term freq value
79:         if word[0] in docFreqDict:
80:             #print(word)
81:
82:             df=docFreqDict[word[0]] # get the document frequency of that word
83:             idf=math.log((numofdocs/df),10) # calculate the idf
84:             tfidf=idf*word[1]  # calculate the tf-idf= tf(word)*idf(word)
85:             dictofwords[word[0]]=tfidf
86:
87:     return dictofwords
88:
89: ## creating a coulum tfidf for each word in an article
90: text_df['tfidf']=text_df.termfrequency.map(temp_tfidf_function)
91:
92: def calculateCosineSimilarity(dict1, dict2):
93:     dotprod=0
94:     for k1 in dict1:
95:         if k1 in dict2.keys():
96:             dotprod=dotprod+(dict1[k1]*dict2[k1])
97:
98:     dict1dot=0
```

```
 99:     dict2dot=0
100:
101:     for key1 in dict1:
102:         dict1dot=dict1dot+(dict1[key1]*dict1[key1])
103:
104:     for key2 in dict2:
105:         dict2dot=dict2dot+(dict2[key2]*dict2[key2])
106:
107:     if(dict1dot == 0 or dict2dot == 0):
108:         cosinesimilarity = 1
109:     else :
110:         cosinesimilarity=dotprod/((math.sqrt(dict1dot))*(math.sqrt(dict2dot)))
111:
112:     return (1 - cosinesimilarity) # we wanted to reverse it to compare it with ja
113:
114: #print('-------------------------------------------------------------------------')
115:
116: #print(text_df['name'])
117: #print(out_link_df['name'])
118:
119: germanJackardListRank = []
120: germanCosineSimilarityListRank = []
121: germanJackardListRank_links = []
122: germanRow = text_df[text_df['name']=='German']
123: germanRow_link = out_link_df[out_link_df['name']=='German']
124: j = 0
125:
126:
127:
128: text_df_rand = pd.DataFrame(columns=text_df.columns)
129: out_link_df_rand = pd.DataFrame(columns=out_link_df.columns)
130:
131: text_df['articleLength']=text_df.text.map(lambda x: len(x))
132: out_link_df['articleLength']=out_link_df.out_links.map(lambda x: len(x))
133: text_df_longest_article=text_df.sort_values(by='articleLength',ascending=[False])
134: out_link_df_longest_article=out_link_df.sort_values(by='articleLength',ascending=
135:
136:
137: for i in range(0,100):
138:     idx = np.random.randint(0, text_df.shape[0])
139:     text_df_rand.loc[i] = text_df.loc[idx]
140:     out_link_df_rand.loc[i] =out_link_df.iloc[idx]
141:
142:
143: rows_count = text_df_rand.shape[0]
144:
145: longestgermanJackardListRank = []
146: longestgermanCosineSimilarityListRank = []
147: longestgermanJackardListRank_links = []
```

```
148:
149: rows_count=100
150:
151: ## for random artciles
152: while (j < rows_count):
153:     row = text_df_rand.iloc[j]
154:     out_link_row = out_link_df_rand.iloc[j]
155:     long_row=text_df_longest_article.iloc[j]
156:     long_out_link_row=out_link_df_longest_article.iloc[j]
157:
158:     start_time = time.time()
159:     germanCosineSimilarityListRank.append(calculateCosineSimilarity(row['tfidf'],
160:     longestgermanCosineSimilarityListRank.append(calculateCosineSimilarity(long_r
161:     end_time = time.time()
162:     germanJackardListRank.append(calcJaccardSimilarity(row['wordset'], germanRow[
163:     longestgermanJackardListRank.append(calcJaccardSimilarity(long_row['wordset']
164:
165:     germanJackardListRank_links.append(calcJaccardSimilarity(out_link_row['out_li
166:     longestgermanJackardListRank_links.append(calcJaccardSimilarity(long_out_link
167:
168:     j = j + 1
169:     #break
170:
171:
172:
173:
174: #rows_count = text_df_rand.shape[0]
175:
176:
177:
178:
179: print('time taken - ',(end_time - start_time))
180:
181: #print(germanCosineSimilarityListRank)
182: #print(germanJackardListRank)
183:
184: def kendalls_tau(list1, list2):
185:     score = 0
186:     con_pair = 0
187:     non_con_pair = 0
188:
189:     length = len(list1)
190:
191:     if(length > len(list2)) :
192:         length = len(list2)
193:
194:     i = 0
195:     while(i < length) :
196:         j = i + 1
```

```
197:         while(j < length) :
198:             if(((list1[i] - list1[j]) * (list2[i] - list2[j])) >= 0):
199:                 con_pair = con_pair + 1
200:             else:
201:                 non_con_pair = non_con_pair + 1
202:             j = j + 1
203:         i += 1
204:
205:     score = (con_pair - non_con_pair) / (con_pair + non_con_pair)
206:     return score
207: print('*** for 100 random articles***')
208: print('kendalls Tau score for text based cosine and jaccard similarity - ', kendal
209: print('kendalls Tau score for text based cosine and link based jaccard similarity
210: print('kendalls Tau score for text based jaccard and link based jaccard similarity
211:
212: print('*** for 100 longest articles***')
213:
214: print('kendalls Tau score for text based cosine and jaccard similarity - ', kendal
215: print('kendalls Tau score for text based cosine and link based jaccard similarity
216: print('kendalls Tau score for text based jaccard and link based jaccard similarity
217:
218:
219: '''
220: for row in text_iterate:
221:     #print(row[0])
222:     #print(row[1].tfidf)
223:     print(type(germanRow['tfidf']))
224:     print(type(row[1]['tfidf']))
225:     germanListRank.append(calculateCosineSimilarity(germanRow['tfidf'], row[1]['t
226:     #print(type(row))
227:     break
228: '''
```

```
1:
2:
3: *** for 100 random articles***
4: kendalls Tau score for text based cosine and jaccard similarity -
   -0.21333333333333335
5: kendalls Tau score for text based cosine and link based jaccard similarity -
   0.7337373737373737
6: kendalls Tau score for text based jaccard and link based jaccard similarity -
   0.8424242424242424
7: *** for 100 longest articles***
8: kendalls Tau score for text based cosine and jaccard similarity -
   -0.16646464646464645
9: kendalls Tau score for text based cosine and link based jaccard similarity -
   0.1898989898989899
10: kendalls Tau score for text based jaccard and link based jaccard similarity -
   0.1898989898989899
```

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`.