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: Tango

1. Mariya Chkalova
   mchkalova@uni-koblenz.de
2. Arsenii Smyrnov
   smyrnov@uni-koblenz.de
3. Simon Schauß
   sschauss@uni-koblenz.de

4. Lukas Härtel
lukashaertel@uni-koblenz.de

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 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`.

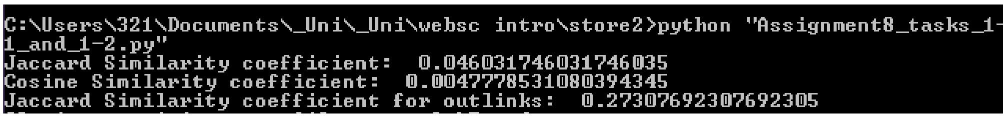
Answer on 1.1 and 1.2 We used the recompiled store.h5 file with ASCII characters only.

Listing 1 Tasks 1.1 and 1.2

```

# coding: utf-8

import pandas as pd
import string
import math

store = pd.HDFStore('store2.h5')
df1=store['df1']
df2=store['df2']
word_dict = dict()
tf_dict = dict()
df_dict = dict()
tfidf_dict = dict()
euclead_dict = dict()
ARTICLES = len(df1)

def create_set(text):
    words = text.lower().split()
    words_pure = set(w.strip(string.punctuation) for w in words)
    return words_pure

def create_dict(index):
    try:
        word_dict[index] = create_set(df1.text.loc[index])
    except:
        print (article_name)
        print (df1[df1.name==article_name].text)


# 1.1.1. 2) Implementation of the function calcJaccardSimilarity

def calcJaccardSimilarity(set1, set2):
    intersection = set1.intersection(set2)
    union = set1.union(set2)
```

```
    return len(intersection)/len(union)

def getId(article):
    idx = df1[df1.name==article].index.tolist()[0]
    return idx

def calc_tf_df(index):
    a_tf_dict = dict()
    text = df1.text.loc[index]
    a_unique_words = word_dict[index]
    for term in a_unique_words:
        tf = text.count(str(term))
        a_tf_dict[term] = tf
        if term in df_dict:
            df_dict[term] += 1
        else:
            df_dict[term] = 1
    tf_dict[index] = a_tf_dict

def calc_tfidf(index):
    a_tfidf = dict()
    a_unique_words = word_dict[index]
    sum_tfidf = 0
    for term in a_unique_words:
        a_tf = tf_dict[index]
        tfidf = a_tf[term] * math.log(ARTICLES/df_dict[term])
        sum_tfidf = sum_tfidf + tfidf * tfidf
        a_tfidf[term] = tfidf
    tfidf_dict[index] = a_tfidf
    euclead_dict[index] = math.sqrt(sum_tfidf)

def calc_scalar(tfIdfDict1, tfIdfDict2):
    wordset1 = set(tfIdfDict1.keys())
    wordset2 = set(tfIdfDict2.keys())
    common_words = wordset1.intersection(wordset2)
    sum_products = 0
    for term in common_words:
        try:
            sum_products = sum_products + tfIdfDict1[term] * tfIdfDict2[term]
        except:
            print (term)
    return sum_products

def calc_Euclead(tfIdfDict):
    coords = list(tfIdfDict.values())
    sum_coords = 0
    for value in coords:
        sum_coords = sum_coords + value * value
```

```
    return math.sqrt(sum_coords)


# 1.1.2 4) Implementation of the fuction calculateCosineSimilarity

def calculateCosineSimilarity(tfIdfDict1, tfIdfDict2):
    return calc_scalar(tfIdfDict1, tfIdfDict2) / (calc_Euclead(tfIdfDict1) * calc_Euclead(tfIdfDict2))


if __name__ == '__main__':

    # 1.1.1. 1) create dictionary that contains set of unique words (value) for each article id (key)

    for index in df1.index.tolist():
        create_dict(index)


    # 1.1.1. 3) Jaccard Similarity for articles Germany, Europe

    Jaccard_sim = calcJaccardSimilarity(word_dict[getId('Germany')], word_dict[getId('Europe')])
    print ('Jaccard Similarity coefficient: ', Jaccard_sim)


    # 1.1.2. 1), 2) populate dictionaries tf_dict and df_dict:
    # tf_dict has article id as a key and a dictionary (term (key) - term frequency (value)) as a value
    # df_dict contains unique terms in all articles as keys and a number of document in which this term is occured as value

    for article_index in df1.index.tolist():
        calc_tf_df(article_index)


    # 1.1.2. 3) populate dictionary tfidf_dict that has article id as a key and a dictionary (term (key) - tfidf (value)) as a value

    for article_index in df1.index.tolist():
        calc_tfidf(article_index)


    # 1.1.2. 5) Cosine Similarity for articles Germany, Europe

    cosine_sim = calculateCosineSimilarity(tfidf_dict[getId('Germany')], tfidf_dict[getId('Europe')])
```



```
C:\Users\321\Documents\_Uni\_Uni\websc intro\store2>python "Assignment8_tasks_1-1_and_1-2.py"
Jaccard Similarity coefficient:  0.04603174603174603 5
Cosine Similarity coefficient:  0.004777853108039434 5
Jaccard Similarity coefficient for outlinks:  0.27307692307692305
```

Figure 1: Similarity coefficients for articles Germany, Europe

```
    print ('Cosine Similarity coefficient: ', cosine_sim)


    # 1.2. Cosine Similarity for out_links of the articles Germany , Europe

    Jaccard_sim_graph = calcJaccardSimilarity(set(df2.loc[getId('Germany')].out_links), set(df2.loc[getId('Europe')].out_links))
    print ('Jaccard Similarity coefficient for outlinks: ', Jaccard_sim_graph)
```

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 The most similar articles will probably not the be same, as the *Jaccard Coefficient* is only taking into account that articles share the same text. They do not perform any clean up on the data, therefore, the range of similarities is influenced by stopwords a lot. Another issue is that count is not taken into the measure, as it is done in *TF-IDF*. The *Jaccard Coefficient on graph* similarity is highly semantic, as it spans across explicitly given links, strongly connected to the represented knowledge; it might be close to the *TF-IDF* approach but that's just a feeling.

Regarding the second question, we want to find out how well the results of the similarity functions fit. A statistical method to find this is the *Residual Sum of Squares* or *RSS*. It could be calculated pairwise on the similarity measures S_1 and S_2, where the function will be $f(a, b)$ the similarity of articles a, b as rated by S_1 and the data-points $x_{a,b}$ the similarity as rated by S_2. The sum would iterate over all $a \in D, b \in D$ where D is the set of all articles.

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?

Listing 2 Tasks 1.4

```
from collections import Counter
from functools import partial, reduce
from itertools import combinations
import pandas as pd
import math
import re
import time


def compose(*functions):
    return lambda args: reduce(lambda acc, f: f(acc), reversed(functions), args)


def split(text):
```

```
    # split text by words
    return re.split(r'\W+', text)


def calcJaccardSimilarity(wordset1, wordset2):
    intersection = len(wordset1 & wordset2)
    union = len(wordset1 | wordset2)
    # return 0 if set union 0
    return 0 if union is 0 else intersection / union


def calcTermFrequency(text):
    return Counter(split(text))


def calcTfIdf(d_term_frequencies, doc_count, wordset, term_frequencies):
    return {term: term_frequencies[term] * math.log(doc_count / d_term_frequencies[term]) for term in wordset}


def flatten(l):
    # http://stackoverflow.com/questions/952914/making-a-flat-list-out-of-list-of-lists-in-python
    return [item for sublist in l for item in sublist]


def calculateCosineSimilarity(tfIdfDict1, tfIdfDict2):
    # gather all terms
    terms = set(tfIdfDict1.keys()) | set(tfIdfDict2.keys())
    # calculate sparse matrix multiplication
    numerator = sum(tfIdfDict1.get(term, 0) * tfIdfDict2.get(term, 0) for term in terms)
    # calculate vector length
    denominator = math.sqrt(sum([v**2for v in tfIdfDict1.values()])) * math.sqrt(
        sum([v**2 for v in tfIdfDict2.values()]))
    return numerator / denominator


with pd.HDFStore('store2.h5', 'r', enconding='utf-8') as store:
    # define sample length
    n = 100

    df2 = store['df2']
    # track rows by name
    df2 = df2.set_index('name')
    # map list of out links to set
    df2['out_links_set'] = df2['out_links'].map(set)
```

```

    df1 = store['df1']
    # count articles
    df1_article_count = len(df1)
    # map text to list of words and then to word set
    df1['word_set'] = df1['text'].map(compose(set, split))
    # calculate overall term frequencies
    document_term_frequencies = Counter(flatten(df1['word_set']))
    # map text to term frequency
    df1['term_frequency'] = df1['text'].map(calcTermFrequency)
    # map wordset with term frequency to tf idf
    df1['tf_idf'] = list(map(lambda t: calcTfIdf(document_term_frequencies, df1_article_count, *t),
                             zip(df1['word_set'], df1['term_frequency'])))
    # map text to text length
    df1['text_length'] = df1['text'].map(len)
    # sort by text length descending and track rows by name
    df1 = df1.sort_values('text_length', ascending=False).set_index('name')
    # initialize counter for computations
    df1_count = 0
    # calculate combinations to calculate
    df1_combination_count = n * (n - 1) / 2
    # n first index
    df1_sets_with_index = df1.index.values[:n + 1]
    # initialize similarity dict
    similarities = {}
    # initialize calculation start time
    start_time = time.time()
    # iterate over pairwise combinations of word sets with index
    for left_index, right_index in combinations(df1_sets_with_index, 2):
        # calculate jaccard similarity for text
        text_jaccard = calcJaccardSimilarity(df1['word_set'].loc[left_index],
                                             df1['word_set'].loc[right_index])
        # calculate cosine similarity for terms
        cosine_similarity = calculateCosineSimilarity(df1['tf_idf'].loc[left_index],
                                                      df1['tf_idf'].loc[right_index])
        # calculate jaccard similarity for out links
        link_jaccard = calcJaccardSimilarity(df2['out_links_set'].loc[left_index],
                                             df2['out_links_set'].
```

```
loc[right_index])
        # store similarity pairs in similarity dictionary
        similarities.update({(left_index, right_index): [text_jaccard, link_jaccard, cosine_similarity]})
        similarities.update({(right_index, left_index): [text_jaccard, link_jaccard, cosine_similarity]})
        # increase computation counter
        df1_count += 1
        # print for feedback for every 100 computations
        if df1_count % 100 is 0:
            print(
                'calculated %i from %i (%f %%)' % (
                    df1_count, df1_combination_count, 100 * df1_count / df1_combination_count))
    print("first %s pairwise similarity calculations: %s seconds" % (n, time.time() - start_time))
```

1. $|computations| = 3 \cdot \frac{|articles| \cdot (|articles|-1)}{2} = 3 \cdot \frac{27493 \cdot (27493-1)}{2} = 1133756334$
2. $\frac{|computations|}{|computations_{first100}|} \cdot time_{first100} = \frac{3 \cdot 377918778}{3 \cdot 4950} \cdot 3.2 seconds = 2d19h$

1.5 Hints:

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

```python
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`.