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.

Group Name : Yankee
Candidates :

1. Sabin Bhattarai - 216203590
   sbhattarai@uni-koblenz.de
2. Biplov K.C. - 216203865
   biplov@uni-koblenz.de
3. Syed Salman Ali. - 216203923
   salmanali@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

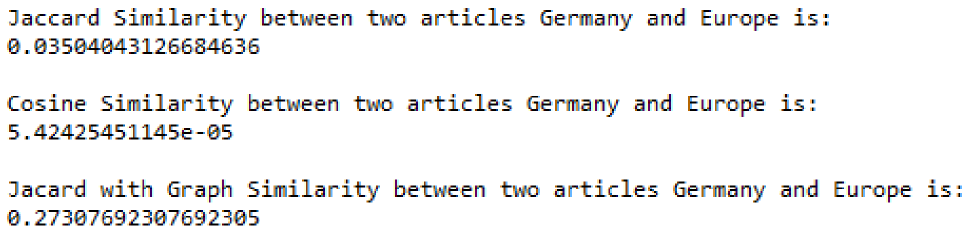
1. Build the word sets of each article for each article id.
2. Implement a function `calcJaccardSimilarity(wordset1, wordset2)` that can calculate the jaccard coefficent of two word sets and return the value.
3. Compute the result for the articles `Germany` and `Europe`.

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`.

Answer

Following is the result obtained from our implemented functions for 1.1.1 and 1.1.2 and 1.2. The code is presented in Answer section of 1.2.



```
Jaccard Similarity between two articles Germany and Europe is:
0.03504043126684636

Cosine Similarity between two articles Germany and Europe is:
5.42425451145e-05

Jacard with Graph Similarity between two articles Germany and Europe is:
0.27307692307692305
```

Figure 1: Obtained results from our implemented cosine, jaccards, jaccard with outlinks

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 1.1.1 and **Answer 1.1.2** and **Answer 1.2** all uses the following code to initially have necessary tfidf calculated, necessary dictionary consisting of articles with respective word list, total word count, total document frequency for each word etc. calculated and saved in files.

Note: The cosine similarity funtion, Jaccard and Jaccard with graph simililarity functions are shown all together in 1.4 answer section.

```

#NOTE: This python file creates necessary files with calculated
# values required for cosine and jackard and statistical measure
# calculation. These files saved heryby will be loaded in next .py file.


import pandas as pd
import itertools
from collections import Counter
import math
import pickle

articles_with_totalwords = {}
articles_list_ofwords = {}
articles_list_ofoutlinks = {}
tf_idf_article_eachterm = {}


# tf_dict has id as article and value = dictionary of each term with associated
# term frequency in that document.
def tf_idf_cal(tf_dict , df_dict):
    global  tf_idf_article_eachterm
    tf_idf_article_eachterm = {}
    total_document_length = len(tf_dict.keys())
    for each_article in tf_dict:
        tf_idf = {}
        for each_term in tf_dict[each_article]:
            tf_idf[each_term]= tf_dict[each_article][each_term] * \
                        math.log(total_document_length / df_dict[each_term],10)
        tf_idf_article_eachterm[each_article] = tf_idf


# returns the dictionary of term frequency
def count_term_frequency_each_article(dict_df1):
   global articles_with_totalwords
   global articles_list_ofwords
   df1_withtermfrequency_eacharticle = {}
   for article_name in dict_df1.keys():
       list_words = dict_df1[article_name][0].split()
       articles_list_ofwords[article_name] = list_words
       articles_with_totalwords[article_name] = len(list_words)
       df1_withtermfrequency_eacharticle[article_name] = \
                 dict(Counter(dict_df1[article_name][0].split()).most_common())
   return df1_withtermfrequency_eacharticle


# updates the unique_term_dict value which contains the all unique terms
# by document frequency for given terms and dataframe
def count_document_frequency_for_eachterm(df1 ,unique_terms_dict ):
   document_frequency_for_eachterm_dic = {}
   copyDF = df1.copy()
   copyDF['wordset']= copyDF.text.map(lambda x: set(x.lower().split()))
   for eacharticleterms in copyDF['wordset']:
        for eachterm in eacharticleterms:
            if eachterm in document_frequency_for_eachterm_dic:
                num=document_frequency_for_eachterm_dic[eachterm]+1
            else:
                num=1
            document_frequency_for_eachterm_dic[eachterm] = num
   return document_frequency_for_eachterm_dic

# Responsible for reading articles from file and evaluating tfIDF , creating
# dictionary of list of words per article and dictionry of article and len words
def read_articles_from_file_evaluate_tfidf(filename):
   global all_terms_basevector_dict
   global articles_list_ofoutlinks
   store = pd.HDFStore(filename)#read .h5 file
   df1=store['df1']
   df2=store['df2']
   df1["text"] = df1.text.str.lower()

   # Dictionary of article names and its associated article text in list form
   dict_df1 = df1.set_index('name').T.to_dict('list')
   dict_df2 = df2.set_index('name').T.to_dict()
   articles_list_ofoutlinks = dict_df2

   # get the term frequency for each article given the dictionary of datas
   articles_with_termfrequency_dict = count_term_frequency_each_article(dict_df1)

   # now we get document frequency . first get all terms and put in the set
   all_terms = list(itertools.chain.from_iterable([list(\
                    term_frequency_dict.keys()) for term_frequency_dict \
                                in articles_with_termfrequency_dict.values()]))
   unique_terms_dict = {each_term : 0 for each_term in all_terms}

   document_frequency_for_eachterm_diction = \
                  count_document_frequency_for_eachterm(df1, unique_terms_dict)

   # Evaluate tfidf for each term in articles. Stored in dictionary
   tf_idf_cal(articles_with_termfrequency_dict , \
                                       document_frequency_for_eachterm_diction)


if __name__ == "__main__":
    # Here we read articles from file, create base vectors, calculate tfidf,
    # calculate document vectors and store all in global variable
    read_articles_from_file_evaluate_tfidf("store.h5")

    # Save  ariticletotalwords, tfidfforeachterm , articles with outlinks
    # and articles with its words into a pickle file.
    try:
        pickle.dump( articles_with_totalwords, open( \
                                            "articles_with_totalwords.p","wb"))
        pickle.dump( tf_idf_article_eachterm, open( \
                                             "tf_idf_article_eachterm.p","wb"))
        pickle.dump( articles_list_ofwords, open( \
                                               "articles_list_ofwords.p","wb"))
        pickle.dump( articles_list_ofoutlinks, open( \
                      "articles_list_ofoutlinks.p","wb"))
    except Exception as e:
        print("ERROR: Failed to save to a file")
        import sys
        sys.exit(1)
```

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:

After the research we decided to use Kendall's Tau as a statistical measure to compare two different ranking measure for the same object.
Kendall's Tau(τ) is a coefficient that represents the degree of concordance between two columns of ranked data.

$$\tau = \frac{\text{number of concordant pairs} - \text{number of discordant pairs}}{\text{number of concordant pairs} + \text{number of discordant pairs}}$$

Concordant pairs: the number of observed ranks below a particular rank which are larger than that particular rank.

Discordant pairs: the number of observed ranks below a particular rank which are smaller in value than that particular rank.

We would perform Kendall's tau and calculate statistical significance on three different combinations: Jaccard co-efficient with cosine similarity, Jaccard co-efficient with Graph with Jaccard co-efficient and Cosine similarity with Graph with Jaccard co-efficient. Below is an example where X could represent cosine similarity and Y could represent Jaccard's co-efficient. Here, X would be the baseline.

X	Y	Concordant	Discordant
1	2	10	1
2	1	10	0
3	4	8	1
4	3	8	0
5	6	6	1
6	5	6	0
7	8	4	1
8	7	4	0
9	10	2	1
10	9	2	0
11	12	0	1
12	11		

Here,
$\tau = \frac{\text{sum of concordant}-\text{sum of discordant}}{\text{sum of concordant}+\text{sum of discordant}}$

or, $\tau = \frac{60-6}{60+6}$

or, $= 0.818$

So, for above given data Kendall's co-efficient is 0.818

Now,

For statistical significance (Z) we use:

$Z = \frac{3*\tau*\sqrt{n(n-1)}}{\sqrt{2(2n+5)}}$
or, $Z = \frac{3*0.818*\sqrt{12(12-1)}}{\sqrt{2(2*12+5)}}$
or, $Z = 3.7019$

For any value of 'Z' if 'Z' is greater than 1.96 the X and Y are statistically significant. After calculating statistical significance for 3 combinations of similarity measure (Jaccard co-efficient with cosine similarity, Jaccard co-efficient - Graph with Jaccard co-efficient and Cosine similarity with Jaccard co-efficient - Graph) on 100 articles- long or random we would plot a bar graph where each segment(Jaccard co-efficient with cosine similarity, Jaccard co-efficient with Graph with Jaccard co-efficient and Cosine similarity with Graph with Jaccard co-efficient) would consists of two bars; one representing number of values less than 1.96 and the other greater than equals to 1.96.

Once the graph is observed, depending upon number of count we could explain if certain article always be the same independent which model we use.

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.

Compute 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 1.4

Note it also has implementations for question part 1.1 and 1.2 in the following code snippet.

1.
The total number of similarity computations that would have to be done can be given as ????????

2.
The total time that would roughly be consumed to do all these computations would be

????????

The following shows the code alongside the plot which implements the above mentioned experiment for statistical measure.

```

import pickle
import numpy as np
import random
from collections import Counter
import operator
import math
import pandas as pd
import matplotlib.pyplot as plt

from_file_articles_with_totalwords = {}
from_file_articles_list_ofwords = {}
from_file_tf_idf_article_eachterm = {}
from_file_articles_list_ofoutlinks = {}

def calculateCosineSimilarity(article1, article2):
    try:
        tf_idf_article1 = from_file_tf_idf_article_eachterm[article1]
        tf_idf_article2 = from_file_tf_idf_article_eachterm[article2]

        scalar_product_article1_article2 = 0
        for each_term_article1 in tf_idf_article1:
            if each_term_article1 in tf_idf_article2.keys():
                scalar_product_article1_article2 = \
                                scalar_product_article1_article2 + \
                                (tf_idf_article1[each_term_article1] * \
                                          tf_idf_article2[each_term_article1])

        euc_dist_tfIdfDict1 = np.sum(np.square(list(tf_idf_article1.values())))
        euc_dist_tfIdfDict2 = np.sum(np.square(list(tf_idf_article2.values())))

        cosineSimilarity = scalar_product_article1_article2 / \
                                    (euc_dist_tfIdfDict1 * euc_dist_tfIdfDict2)

    except Exception as e:
        print("ERROR: Failed to get article for calculating Cosine")
        import sys
        sys.exit(1)
    return cosineSimilarity

def calcJaccardSimilarity(article1, article2):
    wordset1 = from_file_articles_list_ofwords[article1]
    wordset2 = from_file_articles_list_ofwords[article2]
    wordset1=set(wordset1)
    wordset2=set(wordset2)
    inter=wordset1.intersection(wordset2)
    union=wordset1.union(wordset2)
    if (len(union) > 0 ):
        jc=(len(inter)/len(union))
    else:
        jc = 1
    return jc

def calcJaccardGraphSimilarity(article1, article2):
    wordset1 = from_file_articles_list_ofoutlinks[article1]["out_links"]
    wordset2 = from_file_articles_list_ofoutlinks[article2]["out_links"]
    wordset1=set(wordset1)
    wordset2=set(wordset2)
    inter=wordset1.intersection(wordset2)
    union=wordset1.union(wordset2)
    if (len(union) > 0 ):
        jc=(len(inter)/len(union))
    else:
        jc = 1
    return jc



def initialise_loading():
    global from_file_articles_with_totalwords
    global from_file_articles_list_ofwords
    global from_file_tf_idf_article_eachterm
    global from_file_articles_list_ofoutlinks

    # Lets load all necessary already calculated values
    try:

        from_file_articles_with_totalwords = pickle.load(open\
                                          ( "articles_with_totalwords.p","rb"))
        from_file_tf_idf_article_eachterm = pickle.load(open\
                                        ( "tf_idf_article_eachterm.p","rb"))
        from_file_articles_list_ofwords = pickle.load(open\
                                             ( "articles_list_ofwords.p","rb"))
        from_file_articles_list_ofoutlinks = pickle.load(open\
                                          ( "articles_list_ofoutlinks.p","rb"))
    except Exception as e:
        print("ERROR: Failed to load from file")
        import sys
        sys.exit(1)



```

```


#-----------------------------------------------------------------------------
# Begins Experiment for 1.3 and 1.4
#-----------------------------------------------------------------------------

def longest_article(how_many):
    article_in_order = Counter(from_file_articles_with_totalwords).most_common()
    if (len(article_in_order) <= how_many):
        return [article for (article , numberofwords) in article_in_order]
    allarticles = [article for (article , numberofwords) in article_in_order]
    return allarticles[:how_many]

def random_articles(how_many):
    if (len(from_file_articles_with_totalwords) <= how_many):
        return random.sample(from_file_articles_with_totalwords.keys(),len(\
                                          from_file_articles_with_totalwords))
    return random.sample(from_file_articles_with_totalwords.keys(),how_many)

def giverankto_tuplelist(tuple_list):
    i = 1
    ranking_tuple = {}
    for (key , val) in tuple_list:
        ranking_tuple[key] = i
        i = i + 1
    return ranking_tuple


def calculate_statistical_significance(ranked_column):
    C = [sum(i > val for i in ranked_column[idx + 1:]) for idx, val in \
                                                     enumerate(ranked_column)]
    try:
        C.pop()
    except:
        print("No elements to get the Statistic measure\n")


    D = [sum(i < val for i in ranked_column[idx + 1:]) for idx, val in \
                                                      enumerate(ranked_column)]
    try:
        D.pop()
    except:
        print("No elements to get the Statistic measure\n")

    kendalls_tau = (sum(C) - sum(D)) / (sum(C) + sum(D))

    n = len(ranked_column)
    statistical_significance_Z = (3 * kendalls_tau * math.sqrt(n * (n - 1))) / \
                                                         math.sqrt(2 * (2*n+5))

    #Note Any Z value greater than 1.96 is going to statistically significant.
    return statistical_significance_Z


# Cosine vs Jacakard
def get_statistical_significance_cosine_jacakard(firstArticle , \
                                                     remainingArticles_list):
    col1_from_cosine_firstcompared_toothers = {remainingArticles_list[i]: \
           calculateCosineSimilarity(firstArticle , remainingArticles_list[i])\
                                                       for i in range (0,100)}
    sorted_cosine = sorted(col1_from_cosine_firstcompared_toothers.items(), \
                           key=operator.itemgetter(1) , reverse=True)
    ranked_cosine_dict = giverankto_tuplelist(sorted_cosine)


    col2_from_jackard_firstcompared_toothers = {remainingArticles_list[i]: \
           calcJaccardSimilarity(firstArticle , remainingArticles_list[i]) \
                                                       for i in range (0,100)}
    sorted_jacard = sorted(col2_from_jackard_firstcompared_toothers.items(), \
                           key=operator.itemgetter(1) , reverse=True)
    ranked_jacard_column = [ranked_cosine_dict[term] for (term , value) \
                                                             in sorted_jacard]
    return calculate_statistical_significance(ranked_jacard_column)

#Cosine vs Jackard with outlinks - Graphs
def get_statistical_significance_cosine_jacakardGraph(firstArticle , \
                                                     remainingArticles_list):
    col1_from_cosine_firstcompared_toothers = {remainingArticles_list[i]: \
        calculateCosineSimilarity(firstArticle , remainingArticles_list[i]) \
                                                      for i in range (0,100)}
    sorted_cosine = sorted(col1_from_cosine_firstcompared_toothers.items(), \
                                     key=operator.itemgetter(1) , reverse=True)
    ranked_cosine_dict = giverankto_tuplelist(sorted_cosine)


    col2_from_jackardGraph_firstcompared_toothers = {remainingArticles_list[i]:\
        calcJaccardGraphSimilarity(firstArticle , remainingArticles_list[i]) \
                                                       for i in range (0,100)}
    sorted_jacard = sorted(col2_from_jackardGraph_firstcompared_toothers.items(),\
                                     key=operator.itemgetter(1) , reverse=True)
    ranked_jacard_column = [ranked_cosine_dict[term] for (term , value) in \
                                                               sorted_jacard]
    return calculate_statistical_significance(ranked_jacard_column)


# Jackard vs Jackard with outlinks - Graphs
def get_statistical_significance_jacakard_jacakardGraph(firstArticle , \
                                                       remainingArticles_list):
```

```
    col1_from_cosine_firstcompared_toothers = {remainingArticles_list[i]: \
               calcJaccardSimilarity(firstArticle , remainingArticles_list[i])\
                                                        for i in range (0,100)}
    sorted_cosine = sorted(col1_from_cosine_firstcompared_toothers.items(), \
                                     key=operator.itemgetter(1) , reverse=True)
    ranked_cosine_dict = giverankto_tuplelist(sorted_cosine)


    col2_from_jackardGraph_firstcompared_toothers = {remainingArticles_list[i]:\
           calcJaccardGraphSimilarity(firstArticle , remainingArticles_list[i])\
                                                        for i in range (0,100)}
    sorted_jacard = sorted(col2_from_jackardGraph_firstcompared_toothers.items(),
                                     key=operator.itemgetter(1) , reverse=True)
    ranked_jacard_column = [ranked_cosine_dict[term] for (term , value) in \
                                                                 sorted_jacard]
    return calculate_statistical_significance(ranked_jacard_column)


# We will evaluate each pair and do the plot by measuring whish was more
# significant compared to each other via frequency plot
def performStatistical_significance_and_plot(type_of_article_random_or_longest):
    cosine_jackard_dict = {}
    cosine_jackard_graph_dict = {}
    jackard_jackard_graph_dict = {}
    for article in type_of_article_random_or_longest:
        temp = type_of_article_random_or_longest[:]
        temp.remove(article)
        cosine_jackard_dict[article] =  \
                   get_statistical_significance_cosine_jacakard(article , temp)
        cosine_jackard_graph_dict[article] =  \
              get_statistical_significance_cosine_jacakardGraph(article , temp)
        jackard_jackard_graph_dict[article] =  \
            get_statistical_significance_jacakard_jacakardGraph(article , temp)


    # ---------------------------------------------------------------------------
    # Counting the total significance for each pair and plotting
    Cosine_JackardDict = (list(cosine_jackard_dict.values()))
    Cosine_JackardGraphDict = (list(cosine_jackard_graph_dict.values()))
    Jackard_JackardGraphDict = (list(jackard_jackard_graph_dict.values()))

    df = pd.DataFrame(
        {'Cosine_Jackard': Cosine_JackardDict,
         'Cosine_JackardGraph': Cosine_JackardGraphDict,
         'Jackard_JackardGraph': Jackard_JackardGraphDict
        })
    df_seperated = pd.DataFrame({'Insignificant' : df[df<= 1.96].count(),
         'Significant' : df[df > 1.96].count()})
    df_seperated.plot(kind='bar')
    plt.ylabel("Frequency")
    plt.show()


if __name__ == "__main__":

    initialise_loading()
    print("\n Jaccard Similarity between two articles Germany and Europe is: \n"\
                                 , calcJaccardSimilarity("Germany" , "Europe" ))
    print("\n Cosine Similarity between two articles Germany and Europe is: \n"\
                            , calculateCosineSimilarity("Germany" , "Europe" ))
    print("\n Jacard with Graph Similarity between two articles Germany and " + \
             "Europe is: \n" , calcJaccardGraphSimilarity("Germany" , "Europe"))

    #------------------------------------------------------------------------
    # Answer 1.4 Performing Experiment and testing with plots
    #------------------------------------------------------------------------

    # Here I will have to get hundred Longest and random Article comparisons
    hundredlongestArticle = longest_article(101)
    randomarticles = random_articles(101)

    performStatistical_significance_and_plot(hundredlongestArticle)
    performStatistical_significance_and_plot(randomarticles)
```

Strategy 1 : 100 Longest Articles

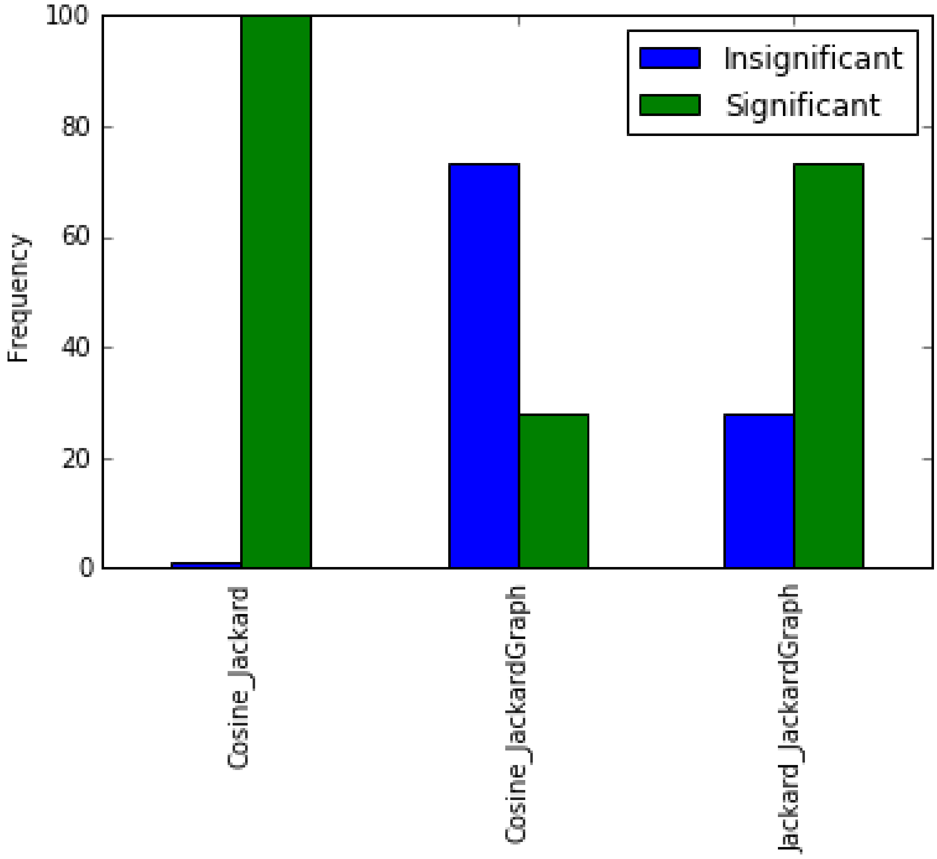


Figure 2: Obtained significance evaluated through frequency measure.

Strategy 2 : 100 Random Articles

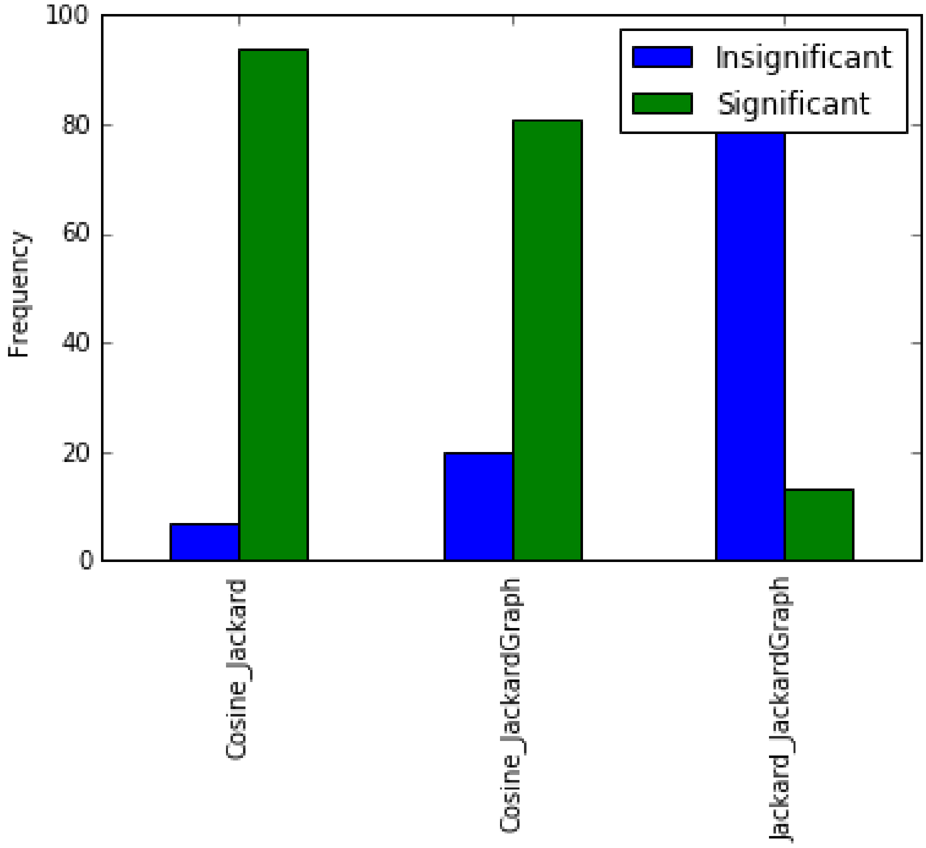


Figure 3: Obtained significance evaluated through frequency measure.

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`.