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: XXXX

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`.

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.

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: Code:

```
# assignment 8
# Andrea Mildes - mildes@uni-koblenz.de
# Sebastian Blei - sblei@uni-koblenz.de
# Johannes Kirchner - jkirchner@uni-koblenz.de
# Abdul Afghan - abdul.afghan@outlook.de

import logging
import time
import pandas as pd
import re
import numpy as np
from numpy import zeros
import operator
np.set_printoptions(threshold=np.nan)


document_freq = {}
word_vector = {}
article_dic = {}
sparse_tfidf_vectors = {}
df1 = pd.DataFrame()
df2 = pd.DataFrame()

# Configure logging and set start time
logging.basicConfig(filename='similarity.log', level=logging.DEBUG)
start_time = 0


# Read the given file into a string
def read_file(file):
    with open(file) as f:
        data = f.read().replace('\n', '')
    return data


# Write a file
def write_file(filename, generated_string):
    with open(filename, 'w') as f:
        f.write(generated_string)

    t = str(round((time.time() - start_time), 2)).zfill(5)
    logging.info("[" + t + "] " +
                 "Finished writing \"" + filename + "\" file. \n")
    return
```

```
45:
46:
47: def create_set(s):
48:     regex = re.compile("\w+")
49:     s_list = regex.findall(s)
50:     word_set = set()
51:
52:     for s in s_list:
53:         word_set.add(s.lower())
54:
55:     return word_set
56:
57:
58: def calc_term_freq(s):
59:     regex = re.compile("\w+")
60:     s_list = regex.findall(s)
61:     word_rank = {}
62:
63:     for s in s_list:
64:         s = s.lower()
65:         try:
66:             # Increase count by one
67:             word_rank[s] += 1
68:         except KeyError:
69:             # Create new dictionary entry for terms that
70:             # are not part of the dict yet
71:             word_rank[s] = 1
72:
73:     return word_rank
74:
75:
76: def calc_document_freq(s):
77:     # create set of the given string in order to get only one
78:     # occurrence of each word
79:
80:     s_set = create_set(s)
81:
82:     for s in s_set:
83:         try:
84:             # Increase count by one
85:             document_freq[s] += 1
86:         except KeyError:
87:             # Create new dictionary entry for chars that
88:             # are not part of the dict. yet
89:             document_freq[s] = 1
90:     return
91:
92:
93: def calcJaccardSimilarity(wordset1, wordset2):
```

```
94:     intersection = wordset1.intersection(wordset2)
95:     union = wordset1.union(wordset2)
96:
97:     jac = len(intersection) / len(union)
98:     return jac
99:
100:
101: # Compute the document frequency and store it in global
102: # dictionary document_freq
103: def log_doc_freq():
104:     t = str(round((time.time() - start_time), 2))
105:     logging.info("[" + t + "]  Compute document frequency ...")
106:
107:     df1.text.apply(calc_document_freq)
108:
109:     # sorted_x = sorted(document_freq.items(), key=operator.itemgetter(1))
110:     # for k, v in sorted_x:
111:     #     print(str(v) + " | " + k)
112:
113:     t = str(round((time.time() - start_time), 2))
114:     logging.info("[" + t + "]  Done! \n")
115:
116:     return
117:
118:
119: # tfidf(word, document) = tf(word, document) * log(|D| / df(word))
120: def calc_tfidf(term, term_freq):
121:     # 1: term frequency of the word in document
122:     # 2: Amount of Documents
123:     amount_of_documents = len(df1)
124:
125:     if len(document_freq) == 0:
126:         logging.error("document_freq dictionary is empty!")
127:         exit()
128:
129:     # 3: document frequency of the term
130:     document_freq_word = document_freq[term]
131:
132:     tfidf = term_freq * np.log(amount_of_documents / document_freq_word)
133:
134:     return tfidf
135:
136:
137: def create_dic_of_all_article_with_terms():
138:     t = str(round((time.time() - start_time), 2))
139:     logging.info("[" + t + "]  Compute tfidf dictionary ...")
140:
141:     dic = {}
142:
```

```
143:     # Iterate over df1
144:     for i in range(0, len(df1)):
145:         articles_dic = {}
146:         word_rank_dic = calc_term_freq(df1.loc[i].text)
147:
148:         # Iterate over the word rank dictionary of the article df1.loc[i]
149:         for k, v in word_rank_dic.items():
150:             # Store each word as key in the article_dic and the
151:             # tfidf of the word as value
152:             articles_dic[k] = calc_tfidf(k, v)
153:
154:         # Store the resulting article_dic as a value and the
155:         # article id as a key in the dic dictionary
156:         dic[i] = articles_dic
157:
158:     t = str(round((time.time() - start_time), 2))
159:     logging.info("[" + t + "]  Done! \n")
160:
161:     return dic
162:
163:
164: # Generate a dictionary with a unique vector for each word
165: def create_word_vectors():
166:     global word_vector
167:
168:     t = str(round((time.time() - start_time), 2))
169:     logging.info("[" + t + "]  Compute word vectors ...")
170:
171:     # Get the amount of all words in all documents
172:     l = len(document_freq)
173:     i = 0
174:
175:     for k, v in document_freq.items():
176:         v = zeros(l)
177:         v[i] = 1
178:         word_vector[k] = v
179:         i += 1
180:
181:     t = str(round((time.time() - start_time), 2))
182:     logging.info("[" + t + "]  Done! \n")
183:
184:     return
185:
186:
187: # Computes the sparse tfidf vector for an article and stores
188: # it in a global dictionary together with its euclidean
189: # length as a tuple
190: # We don't do this in the calculateCosineSimilarity method
191: # because we only want to compute the vector for every article once
```

```
192: # since this improves performance dramatically.
193: def compute_sparse_tfidf_vector(article_id):
194:     global sparse_tfidf_vectors
195:     a = article_dic[article_id]
196:
197:     # Calculate the vector for the given article
198:     article_vector = np.zeros(len(document_freq))
199:     for term, tfidf in a.items():
200:         # Get the vector corresponding to the current word
201:         vec = word_vector[term]
202:         # Multiply vector with tfidf
203:         article_vector += vec * tfidf
204:
205:     sparse_tfidf_vectors[article_id] = (article_vector, np.linalg.norm(article_ve
206:
207:     return
208:
209:
210: # Iterates of a list of article ids and computes its sparse vectors
211: def compute_sparse_tfidf_vector_from_list(article_list):
212:     t = str(round((time.time() - start_time), 2))
213:     logging.info("[" + t + "]  Computing sparse vectors ...")
214:
215:     for i in range(0, len(article_list)):
216:         compute_sparse_tfidf_vector(article_list[i])
217:
218:     t = str(round((time.time() - start_time), 2))
219:     logging.info("[" + t + "]  Done!")
220:
221:     return
222:
223:
224: def calculateCosineSimilarity(tfIdfDict1, tfIdfDict2):
225:     vector1 = tfIdfDict1[0]
226:     vector2 = tfIdfDict2[0]
227:
228:     # Get the dot product of the vectors
229:     dot = vector1.dot(vector2)
230:
231:     # Get the length of both vectors
232:     length_vector1 = tfIdfDict1[1]
233:     length_vector2 = tfIdfDict2[1]
234:
235:     # Some articles are emtpy. If this is the case, mark it
236:     if length_vector1 == 0 or length_vector2 == 0:
237:         return -1
238:
239:     # Inverse cosine of the dot product divided by the product of
240:     # the length of both vectors
```

```
241:     cosine_sim = dot / (length_vector1*length_vector2)
242:
243:     return cosine_sim
244:
245:
246: # Calculate the length of all articles and choose the longest ones
247: def get_longest_articles():
248:     l = []
249:
250:     # Iterate over df1 and append a tuple to the list l
251:     # The tuple contains the index of the text combined with its length
252:     for i in range(0, len(df1)):
253:         l.append((i, len(df1.loc[i].text)))
254:
255:     # Sort the list by text-length descending in place
256:     l.sort(key=lambda tup: tup[1], reverse=True)
257:
258:     # Generate list with 100 entries containing only the article ids
259:     l2 = []
260:     for i in range(0, 100):
261:         l2.append(l[i][0])
262:
263:     return l2
264:
265:
266: # Select 100 random articles
267: def get_random_articles():
268:     l = []
269:
270:     for i in range(0, 100):
271:         l.append(np.random.choice(df1.index, replace=False))
272:
273:     return l
274:
275:
276: def compute_jaccard_similarity_of_all_articles(articles):
277:     t = str(round((time.time() - start_time), 2))
278:     logging.info("[" + t + "]  Compute jaccard similarities "
279:                            "of all articles ...")
280:
281:     matrix = np.zeros(shape=(100, 100))
282:
283:     # Compute every possible combination of articles. We only need to
284:     # calculate the similarity once per pair and we do
285:     # not need to calculate the similarity of the article with itself.
286:     for i in range(0, len(articles)):
287:         i_article = create_set(df1.loc[articles[i]].text)
288:
289:         # Some articles are emtpy. If this is the case, skip it.
```

```
290:         if len(i_article) == 0: continue
291:
292:         for j in range(0, i):
293:             j_article = create_set(df1.loc[articles[j]].text)
294:
295:             # Some articles are emtpy. If this is the case, skip it.
296:             if len(j_article) == 0: continue
297:
298:             jaccard = calcJaccardSimilarity(i_article, j_article)
299:             matrix[i, j] = jaccard
300:
301:         # Logging
302:         if i % 10 == 0:
303:             t = str(round((time.time() - start_time), 2))
304:             logging.info("[" + t + "] Finished " + str(i) + "% of all articles")
305:
306:     t = str(round((time.time() - start_time), 2))
307:     logging.info("[" + t + "]  Done! \n")
308:
309:     return matrix
310:
311:
312: def compute_cosine_similarity_of_all_articles(articles):
313:     t = str(round((time.time() - start_time), 2))
314:     logging.info("[" + t + "]  Compute cosine similarities "
315:                            "of all articles ...")
316:
317:     matrix = np.zeros(shape=(100, 100))
318:
319:     # First compute the sparse vectors for each article in the list
320:     compute_sparse_tfidf_vector_from_list(articles)
321:
322:     # Compute every possible combination of articles. We only need to
323:     # calculate the similarity once per pair and we do
324:     # not need to calculate the similarity of the article with itself.
325:     for i in range(0, len(articles)):
326:         i_vec = sparse_tfidf_vectors[articles[i]]
327:
328:         for j in range(0, i):
329:             j_vec = sparse_tfidf_vectors[articles[j]]
330:
331:             cos_sim = calculateCosineSimilarity(i_vec, j_vec)
332:             # Store the cosine similarity in a matrix
333:             matrix[i, j] = cos_sim
334:
335:         # Logging
336:         if i % 10 == 0:
337:             t = str(round((time.time() - start_time), 2))
338:             logging.info("[" + t + "] Finished " + str(i) + "% of all articles")
```

```
339:
340:     t = str(round((time.time() - start_time), 2))
341:     logging.info("[" + t + "]  Done! \n")
342:
343:     return matrix
344:
345:
346: def compute_jaccard_similarity_for_outlinks_of_all_articles(articles):
347:     t = str(round((time.time() - start_time), 2))
348:     logging.info("[" + t + "]  Compute jaccard similarities for outlinks "
349:                            "of all articles ...")
350:
351:     matrix = np.zeros(shape=(100, 100))
352:
353:     # Compute every possible combination of articles. We only need to
354:     # calculate the similarity once per pair and we do
355:     # not need to calculate the similarity of the article with itself.
356:     for i in range(0, len(articles)):
357:         i_article = set(df2.loc[articles[i]].out_links)
358:
359:         # Some articles are emtpy. If this is the case, skip it.
360:         if len(i_article) == 0: continue
361:
362:         for j in range(0, i):
363:             j_article = set(df2.loc[articles[j]].out_links)
364:
365:             # Some articles are emtpy. If this is the case, skip it.
366:             if len(j_article) == 0: continue
367:
368:             jaccard = calcJaccardSimilarity(i_article, j_article)
369:             matrix[i, j] = jaccard
370:
371:         # Logging
372:         if i % 10 == 0:
373:             t = str(round((time.time() - start_time), 2))
374:             logging.info("[" + t + "] Finished " + str(i) + "% of all articles")
375:
376:     t = str(round((time.time() - start_time), 2))
377:     logging.info("[" + t + "]  Done! \n")
378:
379:     return matrix
380:
381:
382: def pretty_print_matrix(matrix):
383:     for i in range(0, 100):
384:         for j in range(0, 100):
385:             print("%1.3f|" % (matrix[i, j]), end='')
386:         print()
387:
```

```
388:
389: def main():
390:     global start_time, df1, df2, article_dic
391:     # Set start time
392:     start_time = time.time()
393:
394:     # Reset log file
395:     with open('similarity.log', 'w'):
396:         pass
397:
398:     t = str(round((time.time() - start_time), 2))
399:     logging.info("[" + t + "] --- Started --- \n")
400:
401:     store = pd.HDFStore('store2.h5')
402:     df1 = store['df1']
403:     df2 = store['df2']
404:
405:     # ------------------------------------------------#
406:     # 1.1.1 Calculate the Jaccard coefficient for the #
407:     # articles "Germany" and "Europe"                 #
408:     # ------------------------------------------------#
409:     word_set_germany = create_set(
410:         str(df1[df1.name == 'Germany'].text.values[0]))
411:     word_set_europe = create_set(
412:         str(df1[df1.name == 'Europe'].text.values[0]))
413:
414:     print("Jaccard Similarity of Germany and Europe: " +
415:           str(calcJaccardSimilarity(word_set_germany, word_set_europe)))
416:
417:     # ------------------------------------------------#
418:     # 1.1.2 Calculate the cosine similarity for the #
419:     # article "Germany" and "Europe"                #
420:     # ------------------------------------------------#
421:     t = str(round((time.time() - start_time), 2))
422:     logging.info("[" + t + "] --- Started calculations for 1.1.2 ---")
423:
424:     # Compute the document frequency and store it in
425:     # global dictionary document_freq
426:     log_doc_freq()
427:
428:     # Compute a dictionary with the article id as key and
429:     # a dictionary for each article containing the article terms
430:     # and the corresponding tfidf values as value
431:     article_dic = create_dic_of_all_article_with_terms()
432:
433:     # Compute a dictionary to match a unique vector to a specific word.
434:     # This is done before calculating the cosine similarity
435:     # so this only has to be executed once
436:     create_word_vectors()
```

```
437:
438:     # Get ID of the article "Germany" and "Europe"
439:     ger_id = df1[df1.name == "Germany"].index[0]
440:     eur_id = df1[df1.name == "Europe"].index[0]
441:
442:     # Create sparse vectors for each article
443:     compute_sparse_tfidf_vector_from_list([ger_id, eur_id])
444:
445:     # Calculate the cosine similarity of both articles
446:     cos_sim = calculateCosineSimilarity(sparse_tfidf_vectors[ger_id],
447:                                         sparse_tfidf_vectors[eur_id])
448:
449:     print("Cosine Similarity for Germany and Europe: " + str(cos_sim))
450:
451:     t = str(round((time.time() - start_time), 2))
452:     logging.info("[" + t + "] --- Finished calculations for 1.1.2 --- \n")
453:
454:     # -------------------------#
455:     # 1.2 Similarity of graphs  #
456:     # -------------------------#
457:     t = str(round((time.time() - start_time), 2))
458:     logging.info("[" + t + "] --- Started calculations for 1.2 ---")
459:
460:     germany_outlinks = set(df2[df2.name == "Germany"].out_links.values[0])
461:     europe_outlinks = set(df2[df2.name == "Europe"].out_links.values[0])
462:     outlinks_jaccard_sim = calcJaccardSimilarity(germany_outlinks,
463:                                                  europe_outlinks)
464:
465:     print("Jaccard Similarity of outlinks of Germany and Europe: "
466:           + str(outlinks_jaccard_sim))
467:
468:     t = str(round((time.time() - start_time), 2))
469:     logging.info("[" + t + "] --- Finished calculations for 1.2 --- \n")
470:
471:     # -------------------------#
472:     # 1.4 Implement the measure #
473:     # -------------------------#
474:     t = str(round((time.time() - start_time), 2))
475:     logging.info("[" + t + "] --- Started calculations for 1.4 ---")
476:
477:     t = str(round((time.time() - start_time), 2))
478:     logging.info("[" + t + "] --- Started calculations "
479:                            "for the 100 longest articles --- \n")
480:     # Find the 100 longest articles
481:     l_articles = get_longest_articles()
482:
483:     jaccard_matrix = compute_jaccard_similarity_of_all_articles(l_articles)
484:     print("\n\n ---------- JACCARD MATRIX ---------- \n\n")
485:     pretty_print_matrix(jaccard_matrix)
```

```
486:
487:     cosine_matrix = compute_cosine_similarity_of_all_articles(l_articles)
488:     print("\n\n ---------- COSINE MATRIX ---------- \n\n")
489:     pretty_print_matrix(cosine_matrix)
490:
491:     jaccard_outlinks_matrix = compute_jaccard_similarity_for_outlinks_of_all_artic
492:     print("\n\n ---------- JACCARD MATRIX FOR OUTLINKS ---------- \n\n")
493:     pretty_print_matrix(jaccard_outlinks_matrix)
494:
495:     t = str(round((time.time() - start_time), 2))
496:     logging.info("[" + t + "] --- Started calculations "
497:                            "for 100 random articles --- \n")
498:
499:     # Find 100 random articles
500:     r_articles = get_random_articles()
501:     jaccard_matrix = compute_jaccard_similarity_of_all_articles(r_articles)
502:     print("\n\n ---------- JACCARD MATRIX ---------- \n\n")
503:     pretty_print_matrix(jaccard_matrix)
504:
505:     cosine_matrix = compute_cosine_similarity_of_all_articles(r_articles)
506:     print("\n\n ---------- COSINE MATRIX ---------- \n\n")
507:     pretty_print_matrix(cosine_matrix)
508:
509:     jaccard_outlinks_matrix = compute_jaccard_similarity_for_outlinks_of_all_artic
510:     print("\n\n ---------- JACCARD MATRIX FOR OUTLINKS ---------- \n\n")
511:     pretty_print_matrix(jaccard_outlinks_matrix)
512:
513:     t = str(round((time.time() - start_time), 2))
514:     logging.info("[" + t + "] --- Finished calculations for 1.4 --- \n")
515:
516:     t = str(round((time.time() - start_time), 2))
517:     logging.info("[" + t + "] --- Finished ---")
518:
519:     exit(1)
520:
521: if __name__ == '__main__':
522:     main()
```

Word Rank Frequency Diagram:

Answer to 1.3:
The most similar articles to a certain article will be different, depending on which model is used. The reason for this is because the Jaccard Similarity only checks the occurrence of a term in two documents. The amount of occurrences, however, is not being considerd. The Cosine Similarity on the other hand takes these occurrences into account and the similarity rating decreases, the more often the word occurres in a document. This would lead to different similarity rankings, because the Jaccard Similarity would rank documents

with many filler terms higher than the Cosine Similarity, where the rating of those filler terms should be quite low due to the common usage.
One could measure the rankings by setting up a query to a certain subject, as for example a famous person in a certain field, and check if information about this person / subject is ranked higher than information not related to the person / subject. As an example, we take a famous musician and set up a query. The top k most similar articles should be about the person himself, his band, if any, other musicians who play the same instrument or other musicians who played with him.

1.5 Hints:

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

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`.