Yiyi Chen , Informatik plus Computerlinguistik, 11262557

Documentation for Python Project: Concept Linker based on Wikipedia Links

Codes and Input/Output Files

Input Data (size)	Python Code	Output Data (size)
wikilinks_en.txt (5.98G)	aufgabe_1.py	output1.txt (7.44G)
output1.txt (7.44G)	aufgabe_2a.py	linkdist.txt (1.06G)
linkdist.txt (1.06G)	aufgabe_2b.py	linkdist2.txt (24.29M)
linkdist.txt (1.06G)	data_fuer_aufgabe_3.py	wiki2count.csv link2article.csv
wiki2count.csv link2article.csv	combine2dist_fuer_aufgabe_3.py	combine2dist.csv
combine2dist.csv names_line_by_line.txt	aufgabe_3.py	names_line_by_line_result_3.txt
combine2dist.csv	aufgabe_4.py	text_for_linking_result_4.txt
linkdist2.txt (kleinere Linkstatistik)	aufgabe_5a.py	tf.csv (kleinere Linkstatistik) idf.csv(kleinere Linkstatistik)
idf.csv tf.csv (kleinere Statistik)	aufgabe_5b.py	
combine2dist.csv idf.csv tf.csv (kleinere Statistik) names_line_by_line.txt	aufgabe_6.py	names_6.txt

About Codes

Exercise1:

Patterns of the internal links in the data file wikilinks_en.txt are gathered:
(the meaning of the characters are irrelevant, please focus on the syntax)
[[…]] : the box
1.1 [[abc#dfdk|dkfl]]sjkdk (which has a hashtag and a 'l' inside the box, and also has other characters after ']]')
—> abc dkflsjkdk (the result from the pattern 1.1, everything after the hashtag inside the box is removed, and everything before the hashtag is the article name, the concatenation of the characters after 'l' inside the box and the characters after the box is the link text)

1.2 [[abc#dfdk]]sjkdk (which has no 'l' inside inside the box)
—> abc abcsjkdk (the result from the pattern 1.2, everything after the hashtag inside the box is removed, and the everything before the hashtag inside the box is concatenated with everything after]] , which results in the link text)
2.1 [[abc#dfdk|dkfl]] (this pattern differs from the pattern 1.1 in lacking words after the right bracelet ']]')
—> abc dkfl (just removes everything after the hashtag inside the box, and split it using the delimiter 'l')
2.2 [[abc#dfdk]] (no delimiter 'l' , there could be a hashtag which is irrelevant)

Yiyi Chen , Informatik plus Computerlinguistik, 11262557

—> abc abc (just delete everything after the hashtag, and the article name is also the link text)
All the results of the article name and link text is separated by the tab '\t', for the convenience of the data retrieval for the later exercises.
According to the above mentioned patterns, the following methods are created to apply to the data in wikilinks_en.txt leading to the result in output1.txt :

1. delete_frist_bracelet(line) , which deletes the left side bracelet '[[' from the instances which are applied to.
2. delete_last_bracelet(line), which deletes the right side bracelet ']]' from the instances.
3. hashtag_delete(str), which deletes the hashtag and characters after the hashtag.
4. the_first_part(str), which returns the left part of the delimiter 'l'.
5. the_second_part(str), which returns the right part of the delimiter 'l'.
6. is_main_pattern(line), which returns true, when the line is of the most general pattern that includes all above-mentioned patterns.
7. is_pattern_two(line), which returns true, when the line is of the more specific pattern that includes pattern 1.1 and pattern 1.2.

There are other patterns, such as '[[abcl]]' that includes a delimiter 'l' without characters after the delimiter, which doesn't have results. All the patterns are cross checked between the wikilinks_en.txt and *wikipedia internal links*[1].

Exercise2:

a) This exercise aims to calculate the occurrences of the composite of the article name and the link text. The following procedures are operated:

1. The method *FreqDist()* imported from *nltk* module is used for counting the occurrences of each line (the composite of article name and the link text) in the file output1.txt.
2. Splitting each line in the source file with tab '\t' and create a dictionary structured as : { link text : { article name, occurrence of the composite}}.
3. Using the dictionary created from 2, output the data *linkdist.txt* with each line constructed as: occurrence of the composite \t article name \t link text .

b) This exercise aims to render the data structured the same as the data resulted from Exercise 2a), only under the circumstances, that the occurrences of the specific article name is at least 200.

1. With the *defaultdict(data type)* method from *collections* module to create a dictionary from the output file *linkdist.txt*, using the article name as the key and the occurrences of the article name as the value.
2. Looping the line in the file *linkdist.txt* , output the data as the file *linkdist2.txt* on condition of the value of the specific article name in the dictionary created from 1 being bigger than 200.

Occurred problems and solutions:

1. for Exercise 2a), it was very slow to loop in the file *output1.txt* in oder to get the occurrences of the composite of the article name and the link text. Thanks to the advice from Mr. Daniel Weber, using the *update()* method for the dictionary really speeds up the compilation of the program.
2. for Exercise 2b), I was stuck to the idea of making the dictionary structured as {link text :{ article name, occurrence of the article name}}, which did not lead to anywhere. Thanks to the advice from Mr. Benjamin Roth, using a simple split for the line and constructing a *defaultdict(int)* dictionary solved the problem.

Exercise 3:

This exercise 3 aims to return the most frequently occurred article name in terms of the link text. The author found this exercise most disturbing due to the big size of the data *linkdist.txt,* so-called complete link statistic, and the format of the output for the sake of later convenience. Thanks to the advice from Mr. Daniel Weber, the author could construct a dictionary including a tuple as the value constructed as a tuple of the article name and the occurrence of this article name such as :

---

[1] https://en.wikipedia.org/wiki/Wikipedia:Manual_of_Style/Linking#General_principles
https://en.wikipedia.org/wiki/Help:Link

{link text : (article name, occurrence of the article name) } to solve the problem.

1. The dictionary *wiki2count* is constructed from *linkdist.txt* as {article name : occurrence of the article name}, stored in the output file *wiki2count.csv ,* each line occurred as a list [article name, occurrence of the article name].
2. The dictionary link2article, including the values as lists, is constructed from *linkdist.txt* as { link text : [article name1, article name2 …]}, supposing that each link text could be connected with a collection of different article names, stored in the output file *link2article.csv*, each line occurred as a list [link text, [article name1, article name2 ,….]].

1,2 are implemented in the python file *data_fuer_aufgabe_3.py*.

3. In oder to combine the two dictionaries from 1 and 2, the file combine2dist_fuer_aufgabe_3.py is implemented. Looping through two output files and create a new dictionary *combine2dist*, using the link text as the key and the tuple of the article name and its occurrence as the value. Then stored the dictionary in the file *combine2dist.csv*.
4. For the cause of exercise 3, two modes are implemented, the interactive mode and the batch mode.
5. After initiating the interactive mode (without inputing 4 command arguments in the terminal), the user will be asked to input a inquiry to search, if the user hits the ENTER without typing anything, the program terminates. If the user types something and then hit the ENTER key, the search in the dictionary *combine2dist* begins. If the input is found in some link text (the keys) in the dictionary, the corresponding tuples are appended to an empty list. Finally the list of the tuples are compared by the second value of the tuples (the occurrences of the article name), and returns the first value of the tuple (the article name) if its second value is maximal.
6. The batch mode could be initiated by implementing in the terminal, for example, as follows:
   python3 aufgabe_3.py names_line_by_line.txt names_line_by_line_result_3.txt .

Exercise 4:

This exercise aims to combine the Stanford Named Entity Tagger with exercise 3, to obtain the proper names from the input first, and then search these proper names one by one in the link texts in order to get the corresponding most frequently occurred article names and output as a complete sentence.

1. Download and configure the Tagger.[2]
2. Using the Tagger to search the proper names in the input, which would result in a list of tuples containing the the proper name and the tagger. Check if the tagger equals to 'O', when it is not, search the proper name in the *combine2dist*, which retrieved from the output file from exercis3. Appending the corresponding tuples of the article name and its occurrence to an empty list, and then sort the list of tuples by the second value in the tuples to obtain the article name corresponding to the biggest occurrence value. Then replace the construction [[the article namel the proper name]] with the proper name in the original input, and output as the result.
3. The exercise 4 is also implemented in two modes: the interactive mode and the batch mode, as described in Exercise 3 (6).

The Program is experimented with the file *text_for_linking.txt*. The result shows that the result is non relevant to the context of the text. For each proper name in the input, it only gets to pick the most frequently occurred article text, which could be not at all contextually relevant to the input.

Exercise 5:

This exercise aims to use the cosine ranking method to obtain the most probable article in terms of the inquiry.

a) This part renders the data of term frequencies and inverse document frequencies, which are stored individually in the files *tf.csv* and *idf.csv*.

1. One article name could have a collection of different link texts, and one article name is a document. The dictionary *artikeldist* ,using the article name as the key and the collection of the corresponding link texts as the value, is created.

2. In order to prepare the collection of link texts for each article name, the method *inhalts_des_documents(list)* is created. The method is applied to a list because each collection of the link

---

[2] https://sites.google.com/site/rothbenj/teaching/symbolische-programmiersprache/ex_tagging_ranking.pdf?attredirects=0&d=1, see the exercise 1.

texts is a list. This method (1) converts all the characters in the list to lower-case, (2) removes all the punctuations in the list, (3) splits all contents of each link texts with the white spaces, and then (4) converts all the terms resulting from (1),(2),(3) into stems.
3.Then apply the method from 2 to the dictionary from 1 to create a new dictionary *d, constructed as : { {article name1 :{term11, term12, term13.…}, article name 2:{ term21, term22, term23,…}…}.*
4. In order to calculate the *inverse document frequency* of each term, the method *number_of_documents_contain_term(t)* is created, which lopping through all the values of dictionary *d*, to calculate the occurrences of any specific term.
5. The *idf(term)* method ,using the math module, calculate for each term its inverse document frequency, that is *log((Number_of_all_Documents+1)/(number_of_documents_contain_term(term)+1))*, in which the Number_of_all_Documents is simply the size of dictionary *d*'s keys.
6. In order to get the right value for term frequencies, a set of all terms is created, which will not have duplicate elements. Looping through the dictionary *d*'s values, the term set is updated for each loop. Then calculate the *idf(term)* for each term in the set and output to the file *idf.csv*.
The construction of each line in the file: *term \t idf(term*).
7. For tf index, a dictionary *tfdict*, using each article name as the key, and the list of the dictionaries of each term and its tf value as the value, is created, which constructed as : *{ article name1: [{term1: tf1}, {term2: tf2}, ….], article name2 : [{term1: tf1}, {term2: tf2}, …]…}*. Output to the file *tf.csv*.

b) This part uses the data from Exercise 5 a).
1. The method *idf(term)* retrieves the dictionary of each term and its inverse document frequency value.
2. The method *tf()* retrieves the dictionary of each document and its dictionary of terms and term frequencies.
3. The method *inhalt_des_documents(v)* converts the list v into a list of lower-case, punctuations-removed and whitespace-split terms, which will be used to apply to the list of inquiry terms.
4. The method *inquiry_dict(inhalts_der_inqury)* converts the list of inquiry terms resulted from 3 to a dictionary of those terms with its idf*tf* value.
5. The method *document_vector_dict(inhalts_der_inquiry)* applies to the list of the inquiry terms and renders the dictionary of all the relevant documents and the dictionaries of their overlapped terms with their idf*tf values, which works as followed:

5.1 . For each term of the list, if the term is found in the values of the dictionary from 2, then then a dictionary *newdict[(term1, doc)] = tf_term1* is created, and then append this new dictionary to a new list : *documentlst={(term1,doc1):tf_term1,(term2,doc2):tf_term2,…..}*

5.2. Walking through the *documentlst* and looping through the dictionaries of each document, calculate the *idf*tf* value for each term and create the new dictionary from the document and its terms with values (each item with its idf*tf value forms a dictionary), called document_vector_dict:
{document1: [{obama, 0.9838498}, {trump, 0.28392}…..],document2:[{england':2.394934},….]}

5.3. Then all the dictionaries in one document are concatenated, and put the keys and values into a new dictionary, to make sure each item in the values of one document occur only once, the set for the update of the value, constructed as follows:
*newdict= {document1: {term1: idf*tf1, term2: idf*tf2,..}…}*
6. The method *tobevector(dict1,dict2)* convert any two dictionaries into two vectors.
7. The method *cossim(v1,v2)* calculates the cosine similarity between two vectors.
8. The *ranking(inhalts_der_inquiry)* method uses the above-mentioned methods to converts the inquiry and its searched candidates into vectors and compare the cosine similarities, then return the most similar document (wikipedia article name) in terms of the inquiry.

The Exercise 5b) can be implemented in both interactive mode and batch mode. For example, in terminal:
python3 aufgabe_5b.py names_line_by_line.txt names_5b.txt

Problems and Solutions:

Problem1: The compilation raises IndexError: string index out of range because of the importation of PorterStemmer from nltk.stem.porter.
Solution: [3]
In the file porter.py from package nltk/stem/ there is an error in method _ends_double_consonant(), change the method by inserting the if-statement:

---

[3] http://stackoverflow.com/questions/41517595/nltk-stemmer-string-index-out-of-range

```
def _ends_double_consonant(self, word):
    """Implements condition *d from the paper

    Returns True if word ends with a double consonant
    """
    if len(word) < 2:
        return False
    return (
        word[-1] == word[-2] and
        self._is_consonant(word, len(word)-1)
    )
```

Problem2: Have a hard time to retrieve information from txt file and csv file because of the dictionary output.
Solution: Output dictionary as list [k,v] , k is the key of the dictionary and v is the value of the dictionary. Then the output data are very easy to manipulate.

Exercise 6:

This exercise combines the exercise 4 and exercise 5. First, using the StanfordNERTagger to find the proper names from the inquiry; then, finding the most three frequently occurred article names in terms of the link texts; finally, using the methods from exercise 5b to render the most similar article names in terms of the inquiry.

The result from exercise 6 has not been improved compared to the result of exercise 5. The Exercise 6 first uses the method from exercise 4, which can only lead to the non contextually relevant results, and then apply the TF-IDF method with the cosine similarity to get the final result, which would not be better rather worse than simply use the TF-IDF method with the cosine similarity, the result of which renders the most contextually relevant result for this project, even with the partial link statistic.

Problem:

Had a time time to import only several methods from other exercise files without being overwritten by other methods, so the copies of methods rather than import were made.

Observations:

The cosine similarity is better at catching the semantic of each document, the direction the document points can be thought as its meaning so documents with similar meanings will be similar. When modelling documents as vectors, there would be many dimensions which is hard to capture using other distance algorithms such as euclidean, but cosine similarity can capture high dimensional data properly.