Assignment 1

Information Retrieval and Text Mining 20/21

Team Members:

1. Suhas Devendrakeerti Sangolli: (M. No: 3437641)
2. Tushar Rajendra Balihalli: (M. No: 3437638)
3. Stefan Truong: (M. No: 3338287)

Pen and Paper Task 1:

Subtask A:

Given the following documents:

- Document 1: `sun nice`
- Document 2: `sun`
- Document 3: `sun`
- Document 4: `sun`
- Document 5: `water nice`
- Document 6: `water is nice`
- Document 7: `sun sun`
- Document 8: `water`
- Document 9: `water`
- Document 10: `beer`

Let's define the following tables:

Term	docID
sun	1
nice	1
sun	2
sun	3
sun	4
water	5
nice	5
water	6
is	6
nice	6
sun	7
sun	7
water	8
water	9
beer	10

Upon rearrangement

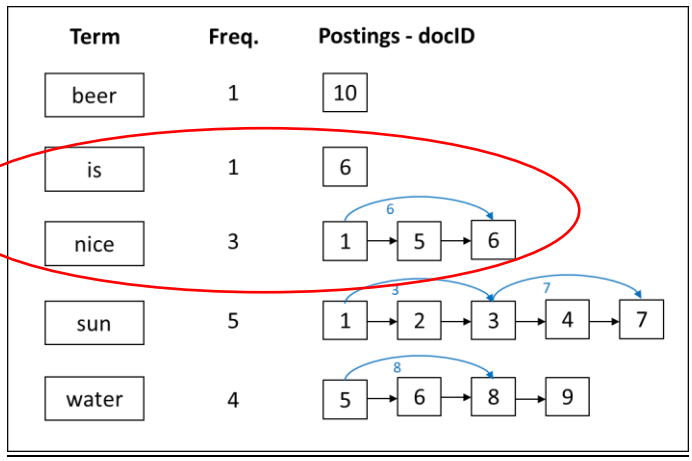
Term	docID
beer	10
is	6
nice	1
nice	5
nice	6
sun	1
sun	2
sun	3
sun	4
sun	7
sun	7
water	5
water	6
water	8
water	9

Then, the inverted index becomes →

Term	Freq.	Postings - docID
beer	1	10
is	1	6
nice	3	1 → 5 → 6
sun	5	1 → 2 → 3 → 4 → 7
water	4	5 → 6 → 8 → 9

Subtask B:

On introduction of Skip pointers, the inverted index looks like below (indicated by blue arrows):



Let's take an example of intersection of words "is" and "nice" in order to illustrate the efficiency of skip pointers easily (encircled red).

Without skip pointers, there would be three comparisons viz. between index "6" and "1", "6" and "5", then "6" and "6". Then it would give a match after the third comparison.

But, on application of skip pointers, initially, the index comparison between "6" and "1" takes place. Since, index "6" is large than index "1", it skips the pointer directly to the next large number "6". Hence, the second comparison itself gives a match and therefore docID "6" is returned as a hit. Here, the number of comparisons reduced to two.

On contrary, as the length of posting lists increases, the comparisons between indexes also increases linearly. Hence, skip pointers allow the query to be answered quickly and efficiently.

Pen and Paper Task 2:

The pseudo code is as follows:

```
import re

def tokenize(text):
    results = []
    for text in text.split(' '):
        results += (re.split('(\W)', text))
    print(results)
```

Explanation: Consider the text "*O'Kennedy hasn't gone to S.*"

Initially, the given text is parsed onto the function defined. We then split the text based on white space only, in order to omit the whitespace character in "for" loop. Next, we split the text in case it doesn't belong to a digit or an alphabet. Implies, symbols like "'" gets split (but not omitted) from each of the text. Hence, on running the above program, we have achieved the following result:

['O', "'", 'Kennedy', 'hasn', "'", 't', 'gone', 'to', 'S', '.']

Linear Run-time: In general, we know that an algorithm is said to have a linear runtime if its running (executing) time increases linearly with the input data, viz. if the input data is of the size 'n', then linear runtime becomes O(n). This defines the time complexity when the algorithm must examine all values of the input data.

To make things simpler, in our example, we just went through all the individual words only once. Hence the algorithm took runtime complexity of 5 units, since the text size is 5 (n=5).

Pen and Paper Task 3:

Given information:

```
term:   (freq.)  docID: <position, position, position, ...>; docID: ...:

Gates:      (4) 1: <3>; 2: <6>;    3: <2,17>; 4: <1>;
IBM:        (2) 4: <3>; 7: <14>;
Microsoft: (4) 1: <1>; 2: <1,21>; 3: <3>;    5: <16,22,51>;
```

For querying: **Gates /2 Microsoft** →

Query Analysis: **The query should return the document ID along with positions**, for which "Gates" is present in same document as "Microsoft", **AND**, where the word "Gates" is present within two places (on either side) of where "Microsoft" is present.

Hence, First, intersection of "Gates" and "Microsoft" is made. Only when the comparison returns a match, then it is further proceeded towards checking the position (as '/k' operator works within same document).

→ Let's analyse the result in real-time for the given example:

At the start, 'docID' of "Gates" and "Microsoft" are compared with very first index '1'. Hence 'docID' gets matched, then it proceeds for the comparison of position index to check for '/2', which also gets matched (Since position '3' of "Gates" is 2 positions away from '1' of "Microsoft"). **Hence, 'docID' = '1' satisfies the query requirements, and is regarded one of the hits.**

Next, 'docID' = '2' gets matched when compared! We proceed with position index. But, on comparing positions, no match is found within 2 places. Hence, this 'docID' doesn't satisfy query requirements and is not regarded as a candidate for hit.

Next, 'docID' = '3' gets matched when compared! Also, on comparing position index, "Gates" is present within 2 positions (indeed, next to) "Microsoft" in docID= 3. **Hence, 'docID' = '3' satisfies the query requirements, and is regarded one of the hits.**

Then, on comparing further docID, there isn't any match found. Hence, the overall query would return the following:

Result →

Gates /2 Microsoft ----------> **1: <3,1>; 3: <2,3>;**

Query: word1 /k word2 docID: <word1_pos, word2_pos>; ….

Pen and Paper Task 4:

For the Damerau-Levenshtein distance, the cells are filled by the following rule:

$$D_{i,j} = \min \begin{cases} D_{i-1,j-1} & +0 \text{ if } u_i = v_j \\ D_{i-1,j-1} & +1 \text{ (replace)} \\ D_{i,j-1} & +1 \text{ (insert)} \\ D_{i-1,j} & +1 \text{ (delete)} \\ D_{i-2,j-2} & +1 \text{ (transpose, if } u_i = v_{j-1} \text{ and } u_{i-1} = v_j) \end{cases}$$

Given the words “who” and “woh”, the Damerau-Levenshtein distance matrix becomes:

		w	**o**	**h**
	0	**1**	**2**	**3**
w	**1**	0	1	2
h	**2**	1	1	1
o	**3**	2	1	1

Referring to the above distance matrix, initialized characters are written in **Bold Black**. Coloured characters (digits) imply distances in general.

Furthermore, if the distance is taken from upper left neighbour (diagonal), then it is colour coded blue. This means, it is either same character match (nothing added), or copy or replace, or transpose operation.

If the distance is taken from upper neighbour, then it is colour coded red. This means we need those many delete operations to be performed.

If the distance is taken from left neighbour, then it is colour coded green. This means we need those many insert operations to be performed.

Cost table for the above words is described below:

cost	**operation**	**input**	**output**
0	copy	w	w
1	transpose	ho	oh

Therefore, we can see that the Damerau-Levenshtein distance between the given two words is “1”, which is to perform one transpose operation.

Programming Task:

1. weiß AND maße

```
Enter term(s) to search: weiß AND maße
Search for "weiß AND maße" using AND in all fields
--------------------------------------------------------------------------------
```

2. weiß AND masse

```
Enter term(s) to search: weiß AND masse
Search for "weiß AND masse" using AND in all fields
--------------------------------------------------------------------------------
```

3. weisse AND maße

```
Enter term(s) to search: weiss AND maße
Search for "weiss AND maße" using AND in all fields
--------------------------------------------------------------------------------
Document_ID 4513
```

4. weisse AND masse

```
Enter term(s) to search: weiss AND masse
Search for "weiss AND masse" using AND in all fields
--------------------------------------------------------------------------------
```

Readme:

Copy the PDF code into the Python IDE. The following libraries are required to run the code.

- pandas
- re
- numpy
- collections

In [4]:
```
import csv
import re
import pandas as pd
import numpy as np
from collections import defaultdict, Counter
def bold(text):
    return '\x1b[1m%s\x1b[0m' % text
def read_CSV():
    '''Reading the input file and storing the id and text of each row as
DATA'''
    df = pd.read_table('postillon.csv', delimiter='\t', names='abcde')
    text_data = df.to_numpy()
    text_data = text_data[1:]
    DATA = []
    for i in range(len(text_data)):
        DATA.append(
            {
                'id':text_data[i][0],
                'description':text_data[i][4]
            }
        )
    return DATA

SPECIAL_CHAR = re.compile(r'[^a-zA-Z0-9]')

def lower(tokens):
    '''The function will change the tokens into lowecase.'''
    for token in tokens:
        yield token.lower()

def tokenize(text):
    '''The function will tokenize the text based on special charectors.
    '''
    yield from SPECIAL_CHAR.split(text)

def find_alpha_numeric(tokens):
    '''The function will return the tokens which contains alpha numeric
s.'''
    for token in tokens:
        if token.isalnum():
            yield token

def merge_and(*arguments):
    '''The function will merge all the documents containing both query t
erm.'''
    if not arguments:
        return Counter()
    output = arguments[0].copy()
    for c in arguments[1:]:
        for doc_id in list(output):
            if doc_id not in c:
                del output[doc_id]
            else:
                output[doc_id] += c[doc_id]
    return output
```

```
def merge_or(*arguments):
    '''The function will merge all the documents containing either of th
e query term.'''
    if not arguments:
        return Counter()
    output = arguments[0].copy()
    for c in arguments[1:]:
        output.update(c)
    return output

def scan(text):
    '''The function will scan for alpha numeric tokens and change the to
kens to lower case.'''
    tokens = tokenize(text)
    for token_filter in (find_alpha_numeric, lower):
        tokens = token_filter(tokens)
    yield from tokens

def index(docs, *fields):
    '''The function will create the inverted index for each term.'''
    index = defaultdict(lambda: defaultdict(Counter))
    for id, d in enumerate(docs):
        for f in fields:
            for t in scan(d[f]):
                index[f][t][id] += 1
    return index

MERGE = {
    'OR': merge_or,
    'AND': merge_and,
}

def search_in_all_fields(index, query, fields):
    '''The function will search the query term and in all the fields and
returns the index of the documents.'''
    for t in scan(query):
        yield MERGE['OR'](*(index[f][t] for f in fields))

def find(index, query, operator='AND', fields=None):
    '''The function will call the function search_in_all_fields with AND
operator as default.'''
    combine = MERGE[operator]
    return combine(*(search_in_all_fields(index, query, fields or index.
keys())))

def query(index, query, operator='AND', fields=None):
    '''The function will print the found documents ids.'''
    print('Search for "%s" using %s in %s' % (bold(query), bold(operator
), fields or 'all fields'))
    print('-'*80)
    ids = find(index, query, operator, fields)
    for doc_id, score in ids.most_common():
        print('Document_ID %s' % bold(DATA[doc_id]['id']))
    print('\n')

DATA = read_CSV()
index = index(DATA, 'id', 'description')
```

```
search_terms = input("Enter term(s) to search: ")
list_terms = search_terms.split(' ')
search_term = " ".join(list_terms)
operator = "AND"
if(len(list_terms)<=1):
    query(index, search_term,operator)
else:
    operator = list_terms[1]
    list_terms.pop(1)
    query(index, search_term,operator)
```

```
Enter term(s) to search: weiss AND masse
Search for "weiss AND masse" using AND in all fields
--------------------------------------------------------------------------
---------
```

In [ ]: