**Assignment 2**

27.11.23

Maria Bruno (3636989), Francesca Carlon (3664579), Linnet Moxon (3387157)

## TASK 1

- What information does the task description contain that the master gives to a Parser?

  It contains a split (subset of documents) for every parser and the task to read a document at a time and create postings lists for the subset

- What information does the parser report back to the master upon completion of the task?

  The parser reports *term-docID-pairs* back to the master sorted into partitions (e.g. a-f, g-p, q-z: each partition for a range of first letters).

- What information does the task description contain that the master gives to an inverter?

  The information contains the task for every inverter to collect all pairs/postings (with *term, docID*) for one term-partition (one partition per inverter: (e.g. for a-f partition), sort them and write them to postings lists.

- What information does the inverter report back to the master upon completion of the task?

  The inverter returns postings list for every term-partition back to the master.

## TASK 2

Subtask 2.1

Subset 1: 10.000 = k*1.000.000^b
Subset 2: 3000 = k*100.000^b

$$\frac{10.000}{3000} = \frac{k * 1.000.000^b}{k * 100.000^b}$$

$$\frac{10}{3} = \frac{1000.000^b}{100.000^b}$$

$$\frac{10}{3} = 10^b$$

$$\log\left(\frac{10}{3}\right) = \log(10^b)$$

$$\log\left(\frac{10}{3}\right) = b \cdot \log(10)$$

$$b = \frac{\log\left(\frac{10}{3}\right)}{\log(10)}$$

into subset 1:

$$10.000 = k * 1.000.000^{\frac{\log\left(\frac{10}{3}\right)}{\log(10)}}$$

$$k = \frac{10.000}{1.000.000^{\frac{\log\left(\frac{10}{3}\right)}{\log(10)}}} = 7,29$$

into subset 2:

$$3000 = k * 100.000^{\frac{\log\left(\frac{10}{3}\right)}{\log 10}}$$

$$k = \frac{3000}{100.000^{\frac{\log\left(\frac{10}{3}\right)}{\log(10)}}} = 7,29$$

$$\Rightarrow b = \frac{\log\left(\frac{10}{3}\right)}{\log(10)} \approx 0.523$$

$$k = 7,29$$

alternative check:

put k and b into subset 2:

$$7,29 * 100.000^{\frac{\log\left(\frac{10}{3}\right)}{\log(10)}} = 3000$$ ✓

Subtask 2.2

Take k and b from subtask 2.1:

k = 7.29
b = log(10/3)/log(10)

M = k*100.000.000^b = 7.29 * (100.000.000^(log(10/3)/log(10))) = 111111,1111 ~ 111.111

The expected vocabulary size for a collection of 100.000.000 tokens is 111.111.

**TASK 3**

variable byte code for the gap 216:

We fill up the bytes with the number starting from the back and fill up the remaining bits in the front with 0:

Binary for 216: 11011000

variable byte code: 00000001 11011000

orange 0 and 1 : continuation bits -> the first one is 0 to indicate that the next byte still belongs to the number 216, the second one is 1, because the number's encoding is finished within that byte.

**TASK 4**

Variable byte code gaps:

10000001 10000010 00000001 11111111 → 00000011111111

3 Numbers
First num = 1
Second number = 2
Third number = 255

Explanation: we divided each byte code gap according to the pattern *continuation bit + 7 bits* and we found three numbers within the provided sequence.

We looked at the first continuation bit in order to decide where the numbers would stop. For

the first one, the continuation bit is 1. This means that after the following 7 digits we will find another number.

The same applies for the second number: its continuation bit is 1, therefore the following 7 digits will represent our second number.

We identified the third number to occupy 7+7 bits (excluding two continuation bits) as follows: the first continuation bit is 0, therefore, the next sequence is part of it. This already gives us a hint that the third number is going to be greater than the previous ones.

Our next step is to identify the position of the 1s in the sequence excluding the continuation bit in case it corresponds to a 1 as well.

So for the first number, the 1 occupies the position of 2 to the power of 0. 2^0 = 1. So the first number is 1.

For the second number, the 1 occupies the position of 2 to the power of 1. 2^1 = 2. So the second number is 2.

For the third number, the 1 occupies the position of:

2^0 = 1
2^1 = 2
2^2 = 4
2^3 = 8
2^4 = 16
2^5 = 32
2^6 = 64
2^7 = 128

Then we summed them up.

So the third number is 255.

**TASK 5**

**with fixed amount of memory:**

9 bytes per term
4 bytes per frequency per term
4 bytes for pointer to postings list per term

14 terms: size of vocabulary

=> (9+4+4)*14 = 17*14 = 238 bytes

**with dictionary as string:**

4 bytes per frequency per term
4 bytes per pointer to postings list
3 bytes per pointer into string
Average: 5.6 bytes per term in string: here we summed the length of each string and divided by the total number of strings.

14 terms: size of vocabulary

=> (4+4+3+5.6)*14 = 232.4 bytes

you save: 238 bytes - 232.4 bytes = 5.6 bytes

**with blocking:**

K = 3 : our block
You save 5 bytes per block
=> 14/3*5 = 14/15 = 0.93 bytes

=> we save 0.93 bytes, so we get 5.6 -0.93 = 4.67 bytes of memory in the end

**PROGRAMMING TASK**

```
import re

def prepare_file(filename):
    # create list to store relevant information
    tweets = []
    # create list to store ID and tokenized and normalized text
    new_content = []
    # read data and store it in a list (line by line)
    with open(filename, encoding="utf8") as f:
        for lines in f:
            line = lines.split('\t')
            line = [line[1], line[4]]
            tweets.append(line)
    # call tokenize_and_normalize to prepare the tweets
    for lst in tweets:
        # new_content.append(lst[0])
        new_content.append(tokenize_and_normalize(lst[1]))
    # return list with IDs and tokenized, normalized tweets
    return new_content
```

```
"""
    helper function for tokenization and normalization
    future normalization to be added: clean for emojis, stop words, web-links
    :return list of tokens in lower case
"""


def tokenize_and_normalize(text):
    normalized_token = []
    # split text by whitespace
    tokens = text.split()
    # normalize all tokens with lower case
    for word in tokens:
        normalized_token.append(word.lower())
    return normalized_token

#print(prepare_file("/home/mabruno/Information_Retrieval/tweets.csv"))


# function that creates a bigram index out of the terms in the tweets
def bigram_index(filename):

    # list of lists containing tokenized and normalized tweets
    tokenized_and_normalized_tweets = prepare_file(filename)

    # dictionary of terms and corresponding bigrams
    term_bigrams_dict = {}

    # list of lists containing the tweets split in bigrams
    processed_tweets_file = []

    # loop that goes through each tokenized tweet
    for tweet in tokenized_and_normalized_tweets:

        # list containing the tweets split in bigrams
        tweet_bigrams = []

        # loop that goes through each token in each tweet
        for term in tweet:

            # list containing bigrams for each term
            term_bigrams = []

            # indices for slicing each term into bigrams
            n = 0
            i = 2
            bigram = ""
```

```
            # add a special character to indicate the end of a term
            term = "$" + term + "$"

            # loop that creates bigrams out of each term
            while i <= len(term):
                bigram = term[n:i]

                #add bigrams to a list of bigrams
                term_bigrams.append(bigram)
                n += 1
                i += 1

            # add bigrams for each term to the tweet list
            tweet_bigrams.append(term_bigrams)
            # example output of each tweet_bigrams:
            # [['he', 'er', 're', 'e$'], ['wh', 'ho', 'o$'], ...]

            # add each term and its bigrams to a dictionary
            term_bigrams_dict.update({term : term_bigrams})
            # example output of each dictionary key and value:
            # {'infection"$': ['in', 'nf', 'fe', 'ec', 'ct', 'ti', 'io', 'on',
'n"', '"$'], ...}

        # list that contains all tweet_bigrams
        # can be returned if needed
        processed_tweets_file.append(tweet_bigrams)

    return term_bigrams_dict

#print(bigram_index("/home/mabruno/Information_Retrieval/tweets.csv"))

# function that takes a queried term and a file
# and searches for the term in that file using a bigram index
def query_bigram_index(queried_term, filename):

    # dictionary having as keys terms found in the given file
    # and as values lists of the rotated corresponding terms
    bigrams_dictionary = bigram_index(filename)

    # lowercase queried term and add end-of-word character
    queried_term = queried_term.lower()
    queried_term = "$" + queried_term + "$"

    # list containing bigrams for the queried term
    queried_term_bigrams = []

    # indices for slicing the term into bigrams
```

```
    n = 0
    i = 2
    bigram = ""

    # loop that creates bigrams out of the queried term
    while i <= len(queried_term):
        bigram = queried_term[n:i]

        #add bigrams to a list of bigrams
        queried_term_bigrams.append(bigram)
        n += 1
        i += 1

    #loop that goes through each bigram of the queried term
    for bigram in queried_term_bigrams:

        # if there is a wildcard in the term
        if ("*" in bigram):

            # remove that bigram from the list of bigrams for that term
            queried_term_bigrams.remove(bigram)


# at this point, the function should loop through the values of the bigrams
dictionary
# and create a new dictionary in which the bigrams are keys
# and the values are lists having as elements the terms containing each bigram
# then it does the intersection of the terms corresponding to each bigram
# by using the set intersection in Python

    # list that will contain the matches found in the bigram index
    # for the bigrams of the queried term
    terms_matching_bigrams = []

    # loop that goes through each key and value of
bigrams_dictionary
    for key_term, bigrams_list in bigrams_dictionary.items():

        # loop that goes through each bigram in the value list
        for bigram in bigrams_list:

            if bigram == queried_term_bigrams:
                terms_matching_bigrams.append(key_term)

    return terms_matching_bigrams

print(query_bigram_index("zeit*","/home/mabruno/Information_Retrieval/tweets.c
sv"))
```

```python
# function that creates a permuterm index out of the terms in the tweets
def permuterm_index(filename):

    # list of lists containing tokenized and normalized tweets
    tokenized_and_normalized_tweets = prepare_file(filename)

    # dictionary of terms and corresponding rotations
    term_rotations_dict = {}

    # loop that goes through each tokenized tweet
    for tweet in tokenized_and_normalized_tweets:

        # loop that goes through each token in each tweet
        for term in tweet:

            # add a special character to indicate the end of a term
            term = term + "$"

            n = len(tweet)

            # list containing the rotations for each term
            rotated_terms = [term]

            # loop that goes through each term and rotates it
            for i in range(n - 1):
                previous_rotation = rotated_terms[-1]
                rotation = previous_rotation[1:] + previous_rotation[0]
                rotated_terms.append(rotation)

            # add each term and its rotations to a dictionary
            term_rotations_dict.update({term : rotated_terms})
            # example output of each dictionary key and value:
            # {'zealand$': ['zealand$', 'ealand$z', 'aland$ze', 'land$zea',
'and$zeal',
            # 'nd$zeala', 'd$zealan', '$zealand', 'zealand$'], ...}

    return term_rotations_dict

#print(permuterm_index("/home/mabruno/Information_Retrieval/tweets.csv"))

# function that takes a queried term and a file
# and searches for the term in that file using a permuterm index
def query_permuterm_index(queried_term, filename):

    # dictionary having as keys terms found in the given file
    # and as values lists of the rotated corresponding terms
    permuterm_dictionary = permuterm_index(filename)
```

```
    # lowercase queried term and add end-of-word character
    queried_term = queried_term.lower()
    queried_term = queried_term + "$"

    #loop that goes through each character of the queried term
    for character in queried_term:

        # if there is a wildcard in the term
        if character == "*":

            # save the index of the character as n
            n = queried_term.index(character)

            # rotate the queried term so that
            # the wildcard is at the right end of the term
            rotated_queried_term = queried_term[n+1:]+queried_term[:n]

            # add wildcard at the end of the rotated queried term
            rotated_queried_term += r"\.*"

    # list that will contain the matches
    # found in the permuterm index for the queried term
    matching_terms = []

    # loop that goes through each key and value of
permuterm_dictionary
    for key_term, rotations_list in permuterm_dictionary.items():

        # loop that goes through each rotated term in the list
        # corresponding to the value in the dictionary
        for rotated_term in rotations_list:

            if rotated_term == rotated_queried_term:
                matching_terms.append(key_term)

    return matching_terms

print(query_permuterm_index("zeit*","/home/mabruno/Information_Retrieval/tweet
s.csv"))
```

Both indexes and query functions help to find terms containing wildcards.
The bigram index is created by taking the tweets file and tokenizing and normalizing the texts of the tweets. Then, each term in each tweet is split in bigrams (after adding a special $ token that helps to indicate the beginning and the end of a term) and the term and its corresponding bigrams (stored in a list) are collected in a dictionary. The terms are the keys and the lists of bigrams are the values of the dictionary. This function is complete and working.
The query function for the bigram index takes a queried term and a file. The file is used to call the bigram index function and create the dictionary of terms and bigrams. This function,

then modifies the queried term so as to split it into bigrams, in the same way as it was done for the terms of the tweets file. The wildcard is removed from the bigrams list corresponding to the queried term. The function is complete and working up to this point. Then, the function should loop through the lists of bigrams from the tweets file and save the terms that contain those bigrams in a set. After that, the intersection should be made between all the terms containing the bigrams of the queried term. That list should postfiltered so as to get rid of false matches.

The permuterm index is created very similarly to the way in which the bigram index is created. The difference is that instead of creating bigrams out of each term in the tweets file, each term is rotated. The rotations for each term are saved in a list. These lists become the values of a dictionary of which the terms are the keys. Also in this case, a $ is added to indicate the end of the term. This function is complete and working.

The query function for the permuterm index is similar to the one for the bigram index. The difference is that the queried term is rotated so as to have the wildcard at the right end of the term. The function is working up to this point. Then, the function should loop through the values of the permuterm index and find rotated terms matching the queried rotated term. When a match is found, the key (the term) corresponding to that match should be saved in a list of matching terms and be returned at the end.

The regular expression useful to find the matches in the permuterm index was not fully implemented. There are two options in that regard: using a regular expression that matches the queried rotated term and any characters after that, namely “.*” or using the caret “^” to match any term in the dictionary having the queried rotated term as the beginning.