Programming Task 1 (10 points)

In the Ilias exercise session you found this PDF in, there is a file available called twit- ter.csv.bz2. Unpack this: bunzip2 twitter.csv.bz2 Each line corresponds to one Tweet. The first column is an id, the second column is the Twitter handle (a user ID), the third column is the name of the user, the fourth column is the Tweet text (with special tokens [NEWLINE] and [TAB]). Implement a method index(filename) which takes the path to the file as an argument and puts all documents into a non-positional inverted index. You can assume that your computer's memory is sufficient to store all postings lists. For the case that it is not, only index a subset of the documents, but it would be better if you think about how to make your program more efficient to make it fit (note that you do not need to store the document text as part of your index). The index should consist of a dictionary and postings lists. Each entry of the dictionary should contain three values: The (normalized) term, the size of the postings list, the pointer to the postings list. Your data structure should be prepared to be able to store the postings lists separately from the dictionary, therefore do not just put a List data structure as value into a HashMap/Tree or Python dictionary. Instead, put the postings lists into a data structure that you could store elsewhere (for instance, use a separate id-to-value mapping for that). It is your decision if and how you normalize tokens and terms. You can also decide to filter out Tweets if you think they are not relevant, to clean the data. Please describe your decisions in your submission. The postings list itself consists of postings which contain each a document id and a pointer to the next postings. For the dictionary, you can use hashing methods included in your programming language (like dictionary in Python or HashMap in Java) or tree structures as available in your programming language (for instance TreeMap in Java). For the postings lists, you can either implement the lists from scratch or use existing data structures (like lists in Python or LinkedList in Java). Then implement a method query(term), where the argument represents one term as a string. It should return the postings list for that term. Then, implement a method query(term1, term2), where you assume that both terms are connected with a logical and. Implement the intersection algorithm as discussed in the lecture for intersecting two postings lists. Do not access the lists array-style (for instance listname[5] where 5 is the position of the element you want to get). Use an iterator (in Python listiter = iter(listname); next(listiter) or in Java iterator.next()). You can choose the programming language. Comment your code! Submit all code in the same PDF as the other tasks (pretty printed). Please note, you won't receive all points if the code is not commented properly. In addition, please query your index for the information need "show me tweets of people who talk about the side effects of malaria vaccines". Provide us with your query and (a subset) of results. The results should be minimally represented by the Tweet-ID and optionally also the Tweet text.

```
from typing import List, Tuple, Dict
import re
import pandas as pd
from nltk.stem import PorterStemmer

class InvertedIndex:
    def __init__(self):
        # stores size of postings list and pointer to list in a tuple
using the normalized term as a key
```

```
        self.dictionary = {}
        # separate data structure which uses the pointer values of the
dict as its keys to get the corresponding posting lists to an entry in
the dict
        self.postings_lists = {}
        # the id counter is used to create new posting lists ids (in
ascending order)
        self.postings_list_id_counter = 0
        self.stemmer = PorterStemmer() # used for stemming in
normalization
        self.dataset: pd.DataFrame = None # optional, used to retrieve
tweet by id

    def normalize_term(self, term: str) -> str:
        """
        Normalize the term by converting it to lowercase, removing any
non-alphanumeric characters, and stemming.
        """
        term = re.sub(r"\W+", "", term.lower())
        # we decided to stem, as it is fast and we do not mind the
risk of overstemming, as the doc size is small therefore wrong results
can fastely be identified by the user.
        stemmed_term = self.stemmer.stem(term)
        return stemmed_term

    def get_tweet_texts(self, tweet_ids: List[str]) -> List[str]:
        '''
        Get the text content of tweets given their IDs.
        '''

        # filter the DataFrame to only include rows with tweet_id in
tweet_ids
        filtered_df =
self.dataset[self.dataset['tweet_id'].isin(tweet_ids)]

        # return the text column of the filtered DataFrame
        return filtered_df['text'].tolist()



    def index(self, filename: str):
        """
        Index the documents in the given file.
        """
        # use quoting = 3 to ignore separators in quotes
        self.dataset = pd.read_csv(filename, sep='\t', header=None,
names=['date', 'tweet_id', 'handle', 'name', 'text'], quoting=3)
        # drop content duplicates if everything except tweet id is
identical
        # why? compresses size and removes redundancy, if texts like
```

```
parols are written multiple times they will be included, as the date
will be different each time
        self.dataset = self.dataset.drop_duplicates(subset=['date',
'handle', 'name', 'text'])
        # sort lines ascending by tweet id, so the postings are
inserted in a sorted way automatically
        self.dataset =
self.dataset.sort_values(by='tweet_id').reset_index(drop=True)
        # one line per tweet
        for _, row in self.dataset.iterrows():
            tweet_id = int(row['tweet_id']) # extract tweet id
            tweet_text = str(row['text']) # extract tweet string
            terms = tweet_text.split() # split on any whitespace char
            unique_terms = set()
            for term in terms:
                # normalize for better query results and less
redundant terms
                normalized_term = self.normalize_term(term)
                if normalized_term and normalized_term not in
unique_terms:
                    unique_terms.add(normalized_term)
                    if normalized_term not in self.dictionary:
                        postings_list_id =
self.postings_list_id_counter
                        # create posting list entry for new term
                        self.postings_lists[postings_list_id] = []
                        # store pointer to posting list in dict
                        self.dictionary[normalized_term] = (0,
postings_list_id)
                        self.postings_list_id_counter += 1

                    # get posting list of normalized term
                    size, postings_list_id =
self.dictionary[normalized_term]
                    postings_list =
self.postings_lists[postings_list_id]
                    # if no postings in list or last posting list
entry does not match id, append the new tweet and let next point to
none
                    if not postings_list or postings_list[-1][0] !=
tweet_id:
                        postings_list.append((tweet_id, None))
                        # update postings list size for term
                        self.dictionary[normalized_term] = (
                            size + 1,
                            postings_list_id,
                        )
                        if len(postings_list) > 1:
                            # update old end-of-postings pointer from
```

```
None to new entry
                            postings_list[-2] = (
                                postings_list[-2][0],
                                len(postings_list) - 1,
                            )

    def query_single_term(self, term: str) -> List[Tuple[int, int]]:
        """
        Query the index for a single term and return the postings
list.
        """
        # normalize query term before checking entries in dict
        normalized_term = self.normalize_term(term)
        if normalized_term in self.dictionary:
            size, postings_list_id = self.dictionary[normalized_term]
            return self.postings_lists[postings_list_id]
        return []

class InvertedIndex(InvertedIndex):
    def intersect_postings_lists(
        self,
        postings_list1: List[Tuple[int, int]],
        postings_list2: List[Tuple[int, int]],
    ) -> List[Tuple[int, int]]:
        """
        Intersect two postings lists and return the common document
IDs.
        """
        result = []
        iter1 = iter(postings_list1)
        iter2 = iter(postings_list2)
        posting1 = next(iter1, None)
        posting2 = next(iter2, None)
        # implementation of two lists as shown in the lecture,
precondition: postings must be sorted in ascending order (was ensured
in index method)
        while posting1 is not None and posting2 is not None:
            doc_id1, next_posting1 = posting1
            doc_id2, next_posting2 = posting2
            if doc_id1 == doc_id2:
                if len(result) > 0:
                    result[-1] = (result[-1][0], len(result))
                result.append((doc_id1, None))
                posting1 = next(iter1, None) if next_posting1 is not
None else None
                posting2 = next(iter2, None) if next_posting2 is not
None else None
            elif doc_id1 < doc_id2:
                posting1 = next(iter1, None) if next_posting1 is not
None else None
```

```
            else:
                posting2 = next(iter2, None) if next_posting2 is not
None else None
        return result


    def query(self, *terms: str) -> List[str]:
        """
        Query the index for any number of AND combined terms and
return the document IDs.
        """
        if not terms:
            return []

        # get postings list for the first term
        postings_list = self.query_single_term(terms[0])

        # intersect postings lists for the remaining terms
        for term in terms[1:]:
            postings_list2 = self.query_single_term(term)
            postings_list =
self.intersect_postings_lists(postings_list, postings_list2)

        # return tweet IDs, without next pointers
        return [doc_id for doc_id, _ in postings_list]

# Create index for tweets file
index = InvertedIndex()
index.index("tweets.csv")

# “show me tweets of people who talk about the side effects of malaria
vaccines”
resp = index.query("malaria", "side", "effects")

print(resp)

index.get_tweet_texts(resp)


# COMMENT: seems to be what we searched for, but if we include vaccine
in the request we will not find these posts as they do not mention
vaccines directly

[968853898185314306, 968853932960251904, 968855540985204738]

['Steroid-based compounds against Malaria: highly effective,
synergistic to artemisinin, no resistance, no side effects, up-scaling
possible. #malaria  https://t.co/minlnwx1f7',
 'Steroid-based compounds against Malaria: highly effective,
synergistic to artemisinin, no resistance, no side...
https://t.co/YDAIkL7jla',
```

```
 'Steroid-based compounds against Malaria: highly effective,
synergistic to artemisinin, no resistance, no side effects, up-scaling
possible. #malaria  https://t.co/mm8ne1EGVS @jlugiessen @GICAfrica
@GSK https://t.co/UQoDej13Uu']
```