In [19]:

```
import sys
import csv
import string
import re
import emoji
import nltk
#nltk.download('stopwords')
from nltk.tokenize import TweetTokenizer
from nltk.corpus import stopwords
from spell_checker import SpellChecker
```

In [20]:

```
class Index:
    """
    This data structure is the value of the indices dictionary.
    """
    def __init__(self, size, pointer2postingsList):
        # size of the postings list

        self.size = size
        # pointer to the head of the postings list
        self.pointer2postingsList = pointer2postingsList
```

In [21]:

```
class PostingNode:
    """
    Linked list for the postings list
    """
    def __init__(self, val):
        self.val = val
        self.next = None
```

In [22]:

```
class SpellChecker(object):
    DEFAULT_DICTIONARIES = {'english': open('englishdic.sec', 'r').read().splitl
ines(),
                            'german': open('germandic-utf8.sec', 'r').read().spl
itlines()}

    def __init__(self, lang_vocab: list, fdist: dict = None, max_edit_distance:
int = 2):
        """
        Creates a `SpellChecker` object from a dictionary and a frequency distri
bution. The principle
        method is `spell_check`. Every other method is called in calling it. To
best understand this
```

```python
        class, start there and work your way through the call sequence: `spell_check` calls
        `candidates` which in turns calls `known` and the edit distance methods.

        :param lang_vocab: a list of words in a language's vocabulary
        :param fdist: a dictionary-like frequency distribution from a large corpus;
                if none is provided and the class default English dictionary is supplied to
                `lang_vocab`, the Brown corpus is imported and used
        :param max_edit_distance: the maximum edit distance at which words will still be considered
        """
        # At present, sticking with the German alphabet even for English
        self.alphabet = 'aäbcdefghijklmnoöpqrsßtuüvwxyz'
        # Key:Value pair of alphabet letters and lists of words beginning with those letters
        # in the provided list of dictionary terms to decrease the time it takes to do dictionary lookups.
        self.lang_vocab = {letter: [word.lower() for word in lang_vocab \
                                    if word.lower().startswith(letter)] for letter in self.alphabet}
        self.fdist = fdist
        self.max_edit_distance = max_edit_distance

        # If no fdist provided and `lang_vocab` is default English, use the Brown news corpus'.
        # In case you don't have a corpus big enough to create a strong frequency distribution
        if self.fdist is None:
            if lang_vocab == SpellChecker.DEFAULT_DICTIONARIES['english']:
                from nltk.corpus import brown
                from nltk import FreqDist

                self.fdist = FreqDist(w.lower() for w in brown.words(categories='news'))

            else:
                raise TypeError('No frequency distribution index provided.')

    def candidates(self, word: str) -> set:
        """
        Returns words within an edit distance of 2 in a ranked order, only generating words
        if there are no results from the previous method. If the word does not begin with
        a letter in `self.alphabet`, it is returned immediately as it was given.
        """
        try:
            return (self.known([word.lower()]) or                      # word if it is known
                    self.known(self.edit_distance1(word.lower())) or   # known words with edit distance 1
```

```
                    self.known(self.edit_distance2(word.lower())) or  # known wo
rds with edit distance 2
                    [word])                                            # word, un
known
        except KeyError:
            return [word]

    def edit_distance1(self, word: str) -> set:
        """
        Thanks to Peter Norvig (http://norvig.com/spell-correct.html)

        Creates all the possible letter combinations that can be made
        with an edit distance of 1 to the word.

        splits = all ways of dividing the word, e.g.
            'word' -> ('w', 'ord'); useful for making changes
        deletions = all ways of removing a single letter, e.g.
            'word'-> 'ord'
        transpositions = all ways of swapping two letters immediately
            adjacent to one another, e.g. 'word' -> 'owrd'
        replacements = all ways of replacing a letter with another
            letter, e.g. 'word' -> 'zord'
        insertions = all ways of inserting a letter at any point in the
            word, e.g. 'word' -> 'wgord'

        :param str word: the relevant word
        :return: a set of terms with an edit distance of 1 from the word
        :rtype: set
        """
        splits = [(word[:i], word[i:]) for i in range(len(word) + 1)]
        deletions = [left + right[1:] for left, right in splits if right]
        transpositions = [left + right[1] + right[0] + right[2:]
                          for left, right in splits if len(right) > 1]
        replacements = [left + letter + right[1:] for left, right
                        in splits if right for letter in self.alphabet]
        insertions = [left + letter + right for left, right in splits
                      for letter in self.alphabet]

        return set(deletions + transpositions + replacements + insertions)

    def edit_distance2(self, word: str) -> set:
        """Simply runs edit_distance1 on every result from edit_distance1(word)"
""
        return set(edit2 for edit in self.edit_distance1(word) for edit2
                   in self.edit_distance1(edit))

    def edit_distanceN(self, word: str) -> set:
        # FIXME
        """Runs `edit_distance1` on the results of `edit_distance1` n times."""
        ret_val = set(word)

        for _ in range(self.max_edit_distance):
            for val in ret_val:
```

```
                ret_val = ret_val | self.edit_distance1(val)


        return ret_val

    def in_dictionary(self, word: str) -> bool:
        """Returns whether the word is in the dictionary."""
        try:
            return word in self.lang_vocab[word[0].lower()]
        except KeyError:
            return False

    def known(self, words: list) -> set:
        """
        Walks through words in a list, checks them against `lang_vocab`,
        and returns a set of those that match.
        """
        return set(w for w in words if len(w) > 1 and self.in_dictionary(w))

    def spell_check(self, word: str) -> str:
        """Chooses the most likely word in a set of candidates based on `word_pr
obability`."""
        return max(self.candidates(word), key=self.word_probability)

    def word_probability(self, word: str) -> int:
        """Divides the frequency of a word by overall token count."""
        try:
            return self.fdist[word.lower()] / len(self.fdist.keys())
        except KeyError:
            return 0
```

In [23]:

```
class TwitterIR(object):
    """
    Main Class for the information retrieval task.
    """
    __slots__ = 'id2doc', 'tokenizer', 'unicodes2remove', 'indices', \
                'urlregex', 'punctuation', 'emojis', 'stop_words', \
                'engSpellCheck', 'gerSpellCheck', 'correctedTerms'

    def __init__(self):
        # the original mapping from the id's to the tweets,
        # which is kept until the end to index the tweets
        self.id2doc = {}
        self.tokenizer = TweetTokenizer()
        # bunch of punctuation unicodes which are not in 'string.punctuation'
        self.unicodes2remove = [
            # all kinds of quotes
            u'\u2018', u'\u2019', u'\u201a', u'\u201b', u'\u201c', \
            u'\u201d', u'\u201e', u'\u201f', u'\u2014',
            # all kinds of hyphens
            u'\u002d', u'\u058a', u'\u05be', u'\u1400', u'\u1806', \
            u'\u2010', u'\u2011', u'\u2012', u'\u2013',
```

```
            u'\u2014', u'\u2015', u'\u2e17', u'\u2e1a', u'\u2e3a', \

            u'\u2e3b', u'\u2e40', u'\u301c', u'\u3030',
            u'\u30a0', u'\ufe31', u'\ufe32', u'\ufe58', u'\ufe63', \
            u'\uff0d', u'\u00b4'
        ]
        # the resulting data structure which has the tokens as keys
        # and the Index objects as values
        self.indices = {}
        # regex to match urls (taken from the web)
        self.urlregex = re.compile('http[s]?://(?:[a-zA-Z]|[0-9]|[$-_@.&+]|[!*\(
\),]'
                                   '|(?:%[0-9a-fA-F][0-9a-fA-F]))+')
        # keep @ to be able to recognize usernames
        self.punctuation = string.punctuation.replace('@', '') + \
                           ''.join(self.unicodes2remove)
        self.punctuation = self.punctuation.replace('#', '')
        self.punctuation = self.punctuation.replace('…', '')
        # a bunch of emoji unicodes
        self.emojis = ''.join(emoji.UNICODE_EMOJI)
        self.emojis = self.emojis.replace('#', '')
        # combined english and german stop words
        self.stop_words = set(stopwords.words('english') + stopwords.words('germ
an'))
        self.engSpellCheck = self._initSpellCheck('english')
        self.gerSpellCheck = self._initSpellCheck('german')
        self.correctedTerms = []    # For demonstration purposes only

    def clean(self, s):
        """
        Normalizes a string (tweet) by removing the urls, punctuation, digits,
        emojis, by putting everything to lowercase and removing the
        stop words. Tokenization is performed aswell.

        :param s the string (tweet) to clean
        :return: returns a list of cleaned tokens
        """
        s = ' '.join(s.replace('[NEWLINE]', '').split())
        s = ' '.join(s.replace('…', '...').split())
        s = self.urlregex.sub('', s).strip()
        s = s.translate(str.maketrans('', '', self.punctuation + string.digits \
                                      + self.emojis)).strip()
        s = ' '.join(s.split())
        s = s.lower()
        s = self.tokenizer.tokenize(s)
        s = [w for w in s if w not in self.stop_words]
        return s

    def _detectLanguage(self, context):
        """
        Detects the language of a tweet based on a hierarchy of criteria:
        1. the number of stopwords from each language in a tweet
        2. the number of "normal" words from each language in a tweet
        3. the more common language, in this case English
```

```
        :param context:

        :return: the determined language of the tweet
        """
        tokens = self.tokenizer.tokenize(context)
        stopsEN = [token for token in tokens if token in stopwords.words('englis
h')]
        stopsDE = [token for token in tokens if token in stopwords.words('german
')]

        # Chooses a language based on the number of stopwords
        if len(stopsEN) > len(stopsDE):
            return 'english'
        elif len(stopsDE) > len(stopsEN):
            return 'german'
        # If that comparison isn't conclusive, it compares the number of words
        # that exist in the respective dictionaries.
        else:
            cleaned = self.clean(context)

            wordsEN = [token for token in cleaned if self.engSpellCheck.in_dicti
onary(token)]
            wordsDE = [token for token in cleaned if self.gerSpellCheck.in_dicti
onary(token)]

            if len(wordsEN) > len(wordsDE):
                return 'english'
            elif len(wordsDE) > len(wordsEN):
                return 'german'
            # If it still cannot decide, it defaults to the more common language
: English
            else:
                return 'english'

    @staticmethod
    def _getGermanFreqDist():
        """
        Walks through germanfreq.txt, splits the values into terms and frequenci
es
        then maps them to a dictionary.

        Ich     489637 -> {ich: 489637}
        ist     475043 -> {ich: 489637, ist: 475043}
        ich     440346 -> {ich: 929983, ist: 475043} - adds to the count of capi
tal 'ich'

        :return: a frequency distribution dictionary of German words
        """
        with open('germanfreq.txt', 'r') as f:
            fdist = {}
            lines = f.read().splitlines()
            for line in lines:
                parts = line.split()
```

```
                # List comprehension is necessary because the file is sometimes
organize "term freq" and sometimes
                # "freq term" and it only works because there are no cardinal nu
mber terms included in the file.
                term = [p for p in parts if not p.isdigit()][0]
                freq = int([p for p in parts if p.isdigit()][0])

                # The entries are split into capital and lowercase; here we comb
ine the two.
                try:
                    fdist[term] = fdist[term] + freq
                except KeyError:
                    fdist[term] = freq

            return fdist

    def _getTokens2ids(self):
        """
        Indexes all the tokens and maps them to a list of tweetIDs.
        :return: a dictionary visualized as {token: [tweetID1, tweetID2, ...]}
        """
        # For the sake of time and presenting functionality, we're limiting the
number
        # of tweets that we are indexing.
        MAX_DOCS_TO_INDEX = 25
        i = 0

        tokens2id = {}

        for id, doc in self.id2doc.items():
            doc = self.clean(doc)
            language = self._detectLanguage(' '.join(doc))

            # This print statement is for demonstration purposes
            print(language, doc)

            for t in doc:
                if language == 'english':
                    # We are specifically excluding handles and hashtags
                    # Nor do we want to spellcheck words that are in the diction
ary
                    if t[0] not in ['@', '#'] and not self.engSpellCheck.in_dict
ionary(t):
                        original = t
                        t = self.spellCheck(t, language)

                        # Collects corrected words for demonstration purposes
                        if original != t:
                            self.correctedTerms.append((original, t))

                elif language == 'german':
                    if t[0] not in ['@', '#'] and not self.gerSpellCheck.in_dict
ionary(t):
```

```
                    original = t

                    t = self.spellCheck(t, language)

                    if original != t:
                        self.correctedTerms.append((original, t))

            if t in tokens2id.keys():
                tokens2id[t].add(id)
            else:
                # a set is used to avoid multiple entries of the same tweetI
D

                tokens2id[t] = {id}

        # Break the loop after MAX_DOCS_TO_INDEX iterations
        i += 1
        if i >= MAX_DOCS_TO_INDEX:
            break

    return tokens2id

def index(self, path):
    """
    1) call the method to read the file in
    2) iterate over the original datastructure id2doc which keeps the mappin
g

    of the tweet ids to the actual tweets and do:
        2a) preprocessing of the tweets
        2b) create a mapping from each token to its postings list (tokens2id
)

    3) iterate over the just created mapping of tokens to their respective
    postings lists (tokens2id) and do:
        3a) calculate the size of the postingslist
        3b) sort the postings list numerically in ascending order
        3c) create a linked list for the postings list
        3d) create the Index object with the size of the postings list and
        the pointer to the postings list - add to the resulting datastructur
e

    :param path: the path to the tweets.csv file
    :return:
    """
    self.initId2doc(path)
    self._indexPostings(self._getTokens2ids())

def _indexPostings(self, tokens2id):
    """
    Creates an `Index` object, which contains a pointer to to the beginning
    of a postings list for every key/token in the `tokens2id` dictionary. It
    stores this in the master inverted index `self.indices`.

    :param tokens2id:
    """
    for t, ids in tokens2id.items():
        # size of the postings list which belongs to token t
```

```
            size = len(ids)

            # sort in ascending order
            ids = sorted(ids)
            # use the first (and smallest) tweetID to be the head node of the
            # linked list
            node = PostingNode(ids[0])
            # keep reference to the head of the linked list since node variable
            # is going to be overridden
            pointer = node
            for id in ids[1:]:
                # create further list items
                n = PostingNode(id)
                # and append to the linked list
                node.next = n
                # step further
                node = n
            # create the index object with size of the postings list
            # and a link to the postings list itself
            i = Index(size, pointer)
            self.indices[t] = i

    def initId2doc(self, path):
        """
        Reads the file in and fills the id2doc datastructure.
        :param path: path to the tweets.csv file
        :return:
        """
        with open(path, 'r', encoding='utf-8', newline='') as f:
            r = csv.reader(f, delimiter='\t')
            for line in r:
                self.id2doc[line[1]] = line[4]
        f.close()

    def _initSpellCheck(self, lang):
        """
        Initializes two `SpellChecker` objects given the path to their dictionar
y files.

        :param lang: the language of the spell checker
        :return: a `SpellChecker` object based on a dictionary in that language
        """

        if lang == 'english':
            # `SpellChecker` will use the Brown FreqDist if none is provided
            freq_dist = None
        elif lang == 'german':
            # For German, we need to create our own.
            freq_dist = self._getGermanFreqDist()
        else:
            raise Exception(f'{lang} is not a supported language.')

        return SpellChecker(SpellChecker.DEFAULT_DICTIONARIES[lang], fdist=freq_
dist)
```

```python
    def intersect(self, pointer1, pointer2):
        """
        Computes the intersection for two postings lists.
        :param pointer1: first postings list
        :param pointer2: second postings list
        :return: returns the intersection
        """
        # create temporary head node
        node = PostingNode('tmp')
        # keep reference to head node
        rvalpointer = node
        while pointer1 and pointer2:
            val1 = pointer1.val
            val2 = pointer2.val
            # only append to the linked list if the values are equal
            if val1 == val2:
                n = PostingNode(val1)
                node.next = n
                node = n
                pointer1 = pointer1.next
                pointer2 = pointer2.next
            # otherwise the postings list with the smaller value
            # at the current index moves one forward
            elif val1 > val2:
                pointer2 = pointer2.next
            elif val1 < val2:
                pointer1 = pointer1.next
        # return from the second element on since the first was the temporary on
e
        return rvalpointer.next

    def _query(self, term, lang):
        """
        Internal method to query for one term.
        :param: term the word which was queried for
        :param: lang the language of the term for spellchecking
        :return: returns the Index object of the corresponding query term
        """
        if lang == 'english':
            if not self.engSpellCheck.in_dictionary(term):
                term = self.spellCheck(term, lang)
        elif lang == 'german':
            if not self.gerSpellCheck.in_dictionary(term):
                term = self.spellCheck(term, lang)

        try:
            return self.indices[term]
        except KeyError:
            return Index(0, PostingNode(''))

    def query(self, *arg):
        """
```

```
        Query method which can take any number of terms as arguments.

        It uses the internal _query method to get the postings lists for the sin
gle
        terms. It calculates the intersection of all postings lists.
        :param *arg term arguments
        :return: returns a list of tweetIDs which all contain the query terms
        """
        language = self._detectLanguage(' '.join([t for t in arg]))
        print(language)  # For demonstration

        # at this point it's a list of Index objects
        pointers = [self._query(t, language) for t in arg if t not in self.stop_
words]
        # here the Index objects get sorted by the size of the
        # postings list they point to
        pointers = sorted(pointers, key=lambda i: i.size)
        # here it becomes a list of pointers to the postings lists
        pointers = [i.pointer2postingsList for i in pointers]
        # first pointer
        intersection = pointers[0]
        # step through the pointers
        for p in pointers[1:]:
            # intersection between the new postings list and the so far
            # computed intersection
            intersection = self.intersect(intersection, p)
            # if at any point the intersection is empty there is
            # no need to continue
            if not intersection:
                return []
        # convert the resulting intersection to a normal list
        rval = []
        pointer = intersection
        while pointer:
            rval.append(pointer.val)
            pointer = pointer.next

        return rval

    def spellCheck(self, term, lang):
        """Runs the relevant spellchecker method."""
        return {'english': self.engSpellCheck,
                'german': self.gerSpellCheck}[lang].spell_check(term)

    def __len__(self):
        """The number of tokens in the inverted index."""
        return len(self.indices.keys())
```

Authors' Note Before We Begin

For Assignment 1 we were docked points beause our postings list was not sorted. We believe this to be a mistake. In the `_indexPostings` methods in the `TwitterIR` class, the `ids` list is sorted before the linked list is created. This occurs in the method's second statement, `sorted(ids)`. Note that in the version of this code that we submitted for Assignment 1, this statement occurs in the `index` method, but for this assignment we created a subroutine for that section of code. Thank you!

Creating the Index

Because the algorithm takes a significant amount of time to run, we have decided to index only the first 25 tweets, which should suffice to demonstrate the `spellcheck` and `_detectLanguage` algorithms.

In the cell below are the 25 tweets represented as lists of tokenized terms, and printed adjacent to them are the results of the language detection algorithm. We are relying — however haphazardly — on the assumption that tweets (and queries) are generally written in a single language, which we then use to spellcheck the content of the tweet.

In [24]:

```
twitterIR = TwitterIR()
twitterIR.index('tweets.csv')
```

```
english ['@knakatani', '@chikonjugular', '@joofford', '@steveblogs',
'says', 'lifetime', 'risk', 'cervical', 'cancer', 'japan', 'means',
'hpv', 'endemic', 'japan', 'screening', 'working', 'well']
english ['@fischerkurt', 'lady', 'whats', 'tumor', '#kippcharts']
german ['@kingsofmetal', 'diagnoseverdacht', 'nunmal', 'schwer', 'ge
rade', 'hausarzt', 'blutbild', 'meist', 'sehen', 'gerade', 'hormone'
, 'überprüft', 'erklärbare', 'gewichtseinlagerungen', 'ja', 'wasser'
, 'fett', 'kind', 'tumor']
german ['@germanletsplay', '@quentin', '@lopoopl', '@levanni', '@ige
loe', '@annelle', 'glückwunsch']
english ['interesting', 'pcr', 'rate', 'major', 'centers', 'authors'
, 'argue', 'treatment', 'compliance', 'major', 'centers', 'see', 'da
tabase', 'think', 'rather', 'due', 'earlier', 'detection', 'smaller'
, 'tumors', 'pcr', 'look', 'deeper', '#crcsm']
english ['new', 'nanobots', 'kill', 'cancerous', 'tumors', 'cutting'
, 'blood', 'supply', '#digitaleconomy', 'february', 'pm']
german ['rip', 'salem', 'aufgrund', 'tumors', 'eingeschläfert']
english ['cancerfighting', 'nanorobots', 'programmed', 'seek', 'dest
roy', 'tumors', 'study', 'shows', 'first', 'applications', 'dna', 'o
rigami', 'nanomedicine', 'nanoroboter', 'schrumpfen', 'tumore', '#me
dtech']
german ['@riptear', 'tumorsdat', 'leg', 'straight', 'mccain']
```

```
english ['quote', 'one', 'statement', 'cancers']
english ['@joyannreid', '@pampylu', 'wrong', 'think', 'trump', 'prob
leme', 'mere', 'small', 'symptom', 'systematic', 'cancer', 'ask', 'a
ntigunkids', 'go', 'look', 'failures', 'us', 'democracy']
english ['erstmal', 'nen', 'anti', 'cancer', 'stick', 'lunge', 'rein
therapieren']
english ['#usa', '#upi', '#news', 'broadcast', '#emetnewspress', 'ob
esity', 'may', 'cause', 'sudden', 'cardiac', 'arrest', 'young', 'peo
ple', 'study', 'says', 'obesity', 'high', 'blood', 'pressure', 'may'
, 'play', 'much', 'greater', 'role', 'sudden', 'cardiac', 'arrest',
'among', 'young', 'people', 'previously', 'thought', 'ne']
english ['leseempfehlung', 'extraordinary', 'correlation', 'obesity'
, 'social', 'inequality']
english ['@fazefuzzface', 'welcome', 'obesity']
english ['@isonnylucas', 'thats', 'exactly', 'point', 'whataboutism'
, 'dont', 'want', 'face', 'problem', 'point', 'worse', 'problem', 'b
following', 'logic', 'yes', 'opioid', 'crisis', 'bad', 'obesity', 'a
ffects', 'way', 'children', 'many', 'deathslike', 'never', 'solve',
'problem']
english ['obese', 'adult', 'free', 'online', 'mobile', 'sex']
english ['obese', 'ebony', 'porn', 'carrie', 'fisher', 'naked', 'pic
tures']
english ['@obesetobeast', 'sad', 'days', 'cant', 'get', 'days', 'stu
ff', 'like', 'reducing', 'damage', 'stuff', 'doesnt', 'move', 'way']
english ['fat', 'obese', 'sex', 'pictures', 'big', 'booty', 'asians'
, 'fucking']
english ['obese', 'porn', 'gallery', 'overcome', 'sex', 'addiction']
german ['amateur', 'downblouse', 'videos', 'get', 'hiv', 'oral', 'se
x']
german ['syphilis', 'kommt', 'wiederja', 'preptherapie', 'hiv', 'übe
rtragungsinfektionsrisiko', 'nahezu', 'null', 'senkenaber', 'syphili
s', 'kleine', 'ficker', 'schon', 'längst', 'dasyphilis', 'tut', 'anf
ang', 'weh', 'ende', 'beißt', 'arsch']
english ['u', 'get', 'hiv', 'oral', 'sex', 'gif', 'teens']
german ['anal', 'sex', 'hiv', 'nudeteenpussyvideos']
```

Corrections

In [25]:

```
for original, corrected in twitterIR.correctedTerms:
        print(f"'{original}' -> '{corrected}'.")
```

```
'hpv' -> 'he'.
'nunmal' -> 'neunmal'.
'pcr' -> 'per'.
'pcr' -> 'per'.
'pm' -> 'am'.
'rip' -> 'trip'.
'nanorobots' -> 'nanobots'.
'applications' -> 'application'.
'tumore' -> 'tumor'.
'cancers' -> 'cancer'.
'probleme' -> 'problem'.
'failures' -> 'failure'.
'erstmal' -> 'resteal'.
'nen' -> 'new'.
'whataboutism' -> 'whatabouts'.
'bfollowing' -> 'following'.
'opioid' -> 'apioid'.
'affects' -> 'affect'.
'children' -> 'chidden'.
'deathslike' -> 'deathlike'.
'online' -> 'unline'.
'porn' -> 'born'.
'pictures' -> 'picture'.
'pictures' -> 'picture'.
'asians' -> 'asian'.
'fucking' -> 'sucking'.
'porn' -> 'born'.
'get' -> 'gut'.
'wiederja' -> 'wieder'.
'senkenaber' -> 'senkender'.
'dasyphilis' -> 'syphilis'.
'hiv' -> 'his'.
```

Testing the Index

German

Below we attempt to query the German terms 'Blutbild' and 'schwer', but by a slip of the finger we accidentally query the non-word 'blutbilt'. The `query` method prints out what it determines to be the language of the query and then prints the relevant TweetIDs. We then fetch the tweet itself and print it for those of us who have not memorized every tweet and its corresponding ID.

In [26]:

```
query_result = twitterIR.query('blutbilt', 'schwer')
print(query_result)
print(twitterIR.id2doc[query_result[0]])
```

```
german
['965695626150326273']
@Kings_of_Metal Ohne Diagnoseverdacht ist es nunmal schwer, gerade f
ür einen Hausarzt. Am Blutbild kann man meist nicht viel sehen, gera
de wenn man nicht auch die Hormone überprüft. Nicht erklärbare Gewic
htseinlagerungen können ja alles sein, von Wasser, Fett, Kind bis hi
n zum Tumor.
```

English

Let's give it an English query: say, all tweets containing the words 'major', 'centers', and 'authors'. But alas! We are bested by the unforgiving cryptology that is English orthography and we instead query 'major', 'senters', and 'authers'. Luckily our spellchecker is there to bail us out.

In [27]:

```
query_result = twitterIR.query('major', 'senters', 'authers')
print(query_result)
print(twitterIR.id2doc[query_result[0]])
```

```
english
['965672579133566980']
Interesting. ⬆️ pCR rate at major centers. Authors argue with ⬆️ tre
atment compliance at major centers. We see the same in our database.
I think it’s rather due to earlier detection, smaller tumors ➡️ more
pCR. Will look deeper into this. #crcsm https://t.co/QfL5g2Z5u9
```

Written Assignments

Task 1

According to cosine similarity, which document di is most relevant to the given query q? Use the log term frequency weight (1 + log10(tf), if tf > 0) as the weight for terms, as discussed in the lecture. What are the values for each comparison? Explain your solution and provide similarity measures for all query document pairs.

- q algorithm intersection
- d1 intersection algorithm for two documents is efficient
- d2 intersection algorithm
- d3 algorithm

Solution

Weight matrix based on log term frequency:

	q algorithm intersection ~~algorithm~~	d1 ~~intersection~~	d2 ~~for~~	d3 ~~two~~	~~documents~~	~~is~~ efficient
algorithm	1	1	1	1		
intersection	1	1	1	0		
for	0	1	0	0		
two	0	1	0	0		
documents	0	1	0	0		
is	0	1	0	0		
efficient	0	1	0	0		

$$\cos(q, d_1) = \frac{q \cdot d_1}{\|q\|_2 \|d_1\|_2} = \frac{(1\cdot 1)+(1\cdot 1)+(0\cdot 1)+(0\cdot 1)+(0\cdot 1)+(0\cdot 1)+(0\cdot 1)}{\sqrt{2}\cdot\sqrt{7}}$$

$$= \frac{2}{4{,}06} = \underline{\underline{0{,}49}}$$

$$\cos(q, d_2) = \frac{q \cdot d_2}{\|q\|_2 \|d_2\|_2} = \frac{(1\cdot 1)+(1\cdot 1)+(0\cdot 0)\dots(0\cdot 0)}{\sqrt{2}\ \sqrt{2}}$$

$$\frac{2}{2{,}83} = 0{,}70$$

$$\cos(q, d_3) = \frac{q \cdot d_3}{\|q\|_2 \|d_3\|_2} = \frac{(1\cdot 1)\dots}{\sqrt{2}\ \sqrt{1}} = \frac{1}{1{,}4} = 0{,}714$$

Task 2

Answer the following questions about distributed indexing:

- What information does the task description contain that the master gives to a parser?
- What information does the parser report back to the master upon completion of the task?
- What information does the task description contain that the master gives to an inverter?
- What information does the inverter report back to the master upon completion of the task?

Solution

- The master tells the parser the location of a split or portion of the corpus on which the parser will work.
- The parser returns a collection of terms paired with their corresponding document identifier, e.g. ('onomatopoeia', 123456789), stored in intermediate segment files, which are partitioned based on an arbitrary quality intrinsic to the terms. For example, one segment file could store terms beginning with letters 'a-f', another 'g-p', and another 'q-z'.
- The master then assigns a machine to be an inverter, which entails walking through a specific class of segment file, say those containing terms beginning with 'a-f'. The inverter's job is to create an index in which each term and a list of the documents it appears in represent a key-value pair.
- The inverter returns all the postings for its assigned term partition.

Task 3

Explain logarithmic merging in your own words. Include the motivation for this method in your explanation and make clear what advantages this method has in contrast to one auxiliary index and only one index on hard disk. How could you use use a distributed compute cluster (for instance with map-reduce) in combination with logarithmic merging? How would you distribute the different merge steps? Which advantages would your solution have and which disadvantages can occur?

Solution

In logarithmic merge, a relatively small index is kept in memory so that common queries can be accomplished without reading from disk. When this auxiliary index becomes too large, it is written to an index on disk. This index has its own size limit, which when reached initiates the creation of a new disk index with a size limit twice as large as the previous one and so and so forth. This minimizes the amount of time spent making expensive merges and allows the indices to dynamically scale with additions to the corpus.

One potential implementation for a distributed compute cluster would be assigning each compute node to be either a parser or inverter, with one lucky and reliable node being crowned the master. An additional set of compute nodes then interefaces with the partitions that the inverters create. Each of these nodes is responsible for holding a partition and carrying out their own logarithmic merge for it. The downside is that each query would need to be split and distributed to the appropriate node.

Task 4

Heaps' law is an empirical law. -- See PDF for collection properties. --

- K means kilo: times 1000
- M means mega: times 1000000
- G means giga: times 1000000000

Task 4.1

Compute the coefficients k and b.

Solution

k ≈ .5 b ≈ 30

Task 4.2

Compute the expected vocabulary size for the complete collection (1G tokens).

Solution

M ≈ 1,000,000

Task 5

Calculate the variable byte code and the gamma code for 217.

Solution

- bin(217) = 11011001
- VB(217) = 00000001 11011001
- γ(217) = 111111101011001

Task 6

From the following sequence of γ-coded gaps, reconstruct first the gap sequence and then the postings sequence: `1111010000111110101011100 0`

Solution

- Gap sequence: [24, 1, 53, 4]
- Posting sequence: [24, 25, 78, 82]

Task 7

Describe in your own words: What is the advantage of the k-gram index vs. the permuterm index for handling wildcard queries? In general, would you take into account properties of the language for the decision which of the two approaches you prefer? Please explain!

Solution

The k-gram's principle advantage is that is does not suffer from the permuterm index's principle disadvantage, which is that a permuted index increases the size of the lexicon by several orders of magnitude. This phenomenon is exacerbated as the size of a lexicon increases. As such, lexicons that are larger in number or average word length are even further handicapped than smaller ones. The k-gram index does, however, have the disadvantage of having a very high recall but low precision because it returns more documents than are actually needed. The trade-off here is storage for accuracy.