K-gram indices

This week the exercise focuses on k-gram indices, and two uses for them: (i) wildcard queries (some*e) and (ii) spell correction (smoetime → sometime).

In [ ]:
```python
import re
import glob
from textutils import tokenize_document
from queryparser import parse_query, ast_has_operation, process_ast, Operation, ParseError
```

The provided window function returns sliding-n-window subsequences of the input sequence:

In [ ]:
```python
def window(seq, n):
    for i in range(len(seq) - n + 1):
        yield seq[i:i+n]
print(list(window([2, 3, 4, 5, 6], 2)))
print(list(window("astrings", 3)))
```

In [ ]:
```python
# Map document titles to document ids
documents = {}
# A running counter for assigning numerical IDs to documents'''
docid_counter = 1
# The posting lists
the_index = {}
# K-gram index
kgram_index = dict()
# The K in k-gram
K = 3
```

Note that from this week onwards the posting lists will be a python set: this way we can use its built in deduplication, & (intersection), | (union), and - (difference):

In [ ]:
```python
eample_set = {3, 4}
eample_set.add(6)
eample_set.add(4)
print(eample_set)
```

Add documents to the index, and words to the k-gram index. Make sure the posting lists are python sets.

For the k-gram index you will want to bracket each term in the vocabulary of your standard inverted index with dollar signs, which are used as word boundary markers and are important to process queries where the wildcard is at the beginning or at the end of a term.
In [ ]:

# wipe existing data
documents = {}
docid_counter = 1
the_index = {}
kgram_index = dict()

for doc in glob.glob('./shared/corpus/*.txt'):
    docid = docid_counter
documents[docid] = doc
docid_counter += 1
print("Added document %s with id %d" % (doc, docid))
for word in tokenize_document(doc):
    the_index.setdefault(word, set()).add(docid)

for word in the_index.keys():
    ### TODO for the assignment: add words to the k-gram index
    pass

Here's a test to check the k-gram index is valid:

In [ ]:

assert(kgram_index['$ze'] == {'zeal', 'zeale', 'zealous', 'zeals', 'zeal-', 'zed'})

Wildcard queries

Write a function that takes a wildcard query term ( `some*e` ) and returns all the k-grams to query in the k-gram index.

In [ ]:

def wildcard_parse(q):
    ### TODO for the assignment: parse the wildcard term and return a list of k-grams
    return []

Tests:

In [ ]:

assert(wildcard_parse('some*e') == ['$so', 'som', 'ome'])
assert(wildcard_parse('*where') == ['whe', 'her', 'ere', 're$'])
assert(wildcard_parse('some*ere') == ['$so', 'som', 'ome', 'ere', 're$'])

Implement querying the k-gram index for matching words, given a certain wildcard query \( q \). Here's an high-level description for the implementation:

- parse the wildcard query with \( \text{wildcard\_parse} \);
- check whether any of the returned kgrams are not in the \( \text{kgram\_index} \); if that's the case, there are no matches for this wildcard query, and and empty set should be returned;
- (optionally) compute the number of matches for each kgram and order them from smaller to larger;
- intersect the word matches for each kgram;
- perform post-processing to exclude false positives (we provide some code that converts the wildcard query in a python regex that only matches valid words);
• return the set of matching words.

In [ ]:

```python
def kgram_wildcard_query(q):
    '''
    Query the k-gram index for words matching q. Return the matches as a set
    '''
    grams = wildcard_parse(q)
    # TODO for the assignment: execute the wildcard query on the k-gram index
    kgram_matches = {}
    post_filter = re.compile('^' + q.replace('*', '\w*') + '$')
    res = {r for r in kgram_matches if post_filter.match(r) is not None}
    return res
```

Don’t forget to perform post-processing to exclude false positives. Tests:

In [ ]:

```python
assert(kgram_wildcard_query('some*e') == {'someone', 'somewhere', 'sometime'})
assert(kgram_wildcard_query('*where') == {'otherwhere', 'everywhere', 'nowhere', 'e
```

We provide a generic implementation of `execute_query`. Note that intersection/union operations are implemented with Python's sets. This implementation will invoke `kgram_wildcard_query` when it encounters a wildcard query during preprocessing. It will replace the wildcard term with a disjunction of the matching words (some*e → someone OR somewhere OR sometime).
In [ ]:

```python
# temporary empty implementation of `spellcorrect`: this will be relevant later

def spellcorrect(arg):
    return None

def execute_query(query):
    def negate(postings):
        return set(documents.keys()) - postings

try:
    ast = parse_query(query)
except ParseException as e:
    print("Failed to parse query '%s'
    % query, e)
    return None

flat = process_ast(ast)

def preprocess_query_tree(tree):
    if tree.op == 'LOOKUP':
        tree.op = 'AND'
    new_args = []
    for arg in tree.args:
        if isinstance(arg, Operation):
            new_args.append(preprocess_query_tree(arg))
        elif '*' in arg:
            kgram_matches = kgram_wildcard_query(arg)
            if len(kgram_matches) == 0:
                print("NOTE: spell-correcting term '%s' because no document contains it")
            elif arg.startswith('-'):
                not_op = Operation('NOT', [Operation('OR', list(kgram_matches))])
                new_args.append(not_op)
        elif tree.op == 'OR':
            new_args.extend(kgram_matches)
        else:
            new_args.append(Operation('OR', kgram_matches))
    else:
        if not arg.startswith('-') and arg not in the_index:
            print("NOTE: spell-correcting term '%s' because no document contains it")
        res = spellcorrect(arg)
        # NOTE: spellcorrect is the second part of this exercise, and will be relevant later
        if res != None and res != []:
            print("NOTE: spell-corrections with Jaccard_threshold for '%s'
            % list(res))
        else:
            new_args.append(arg)

    tree.args = new_args

preprocess_query_tree(flat)

def execute_query_tree(tree):
    result = set()
    if tree.op == 'AND':
        result = set(documents.keys())
    for arg in tree.args:
        if isinstance(arg, Operation):
            temp = execute_query_tree(arg)
        elif arg.startswith('-'):
            temp = negate(the_index[arg[1:]])
        else:
```

temp = the_index[arg]

if tree.op == 'OR':
    result = result | temp
elif tree.op == 'AND':
    result = result & temp
elif tree.op == 'NOT':
    assert(len(tree.args) == 1)
    result = negate(temp)
return result

return execute_query_tree(flat)

In [ ]:

assert(execute_query('some*e') == {1, 3, 4, 5, 6, 8, 10, 11, 12, 13, 14, 15, 16, 18, 23})
assert(execute_query('some*e AND Romeo') == {23})

Spell-checking with a k-gram index

In [ ]:

Jaccard_threshold = .3

Implement spell-correction for a word that was not found in the index, using the k-gram index.

- Compute the k-grams from the input word;
- find the matching words for each k-gram;
- for each candidate word, check the Jaccard coefficient between the input word and the candidate;
- if it's above the provided threshold, add it to the list of

In [ ]:

def spellcorrect(word):
    word_grams = set(window(word, 3))
    ### TODO for the assignment: implement spell correction
    return []

Tests:

In [ ]:

assert(set(spellcorrect('smoetime')) == {'betime', 'betimes', 'Betimes', 'sometime', 'smoetime', 'smoetimes', 'smoetime', 'smoet ime'})

You can try your code as part of the query processor, which already calls spellcorrect whenever a (non-negated) term cannot be found in the index.

In [ ]:

assert(execute_query('smoetime AND Romeo') == {23})