Assignment 3

CS 734: Introduction to Information Retrieval

Fall 2017

Grant Atkins

Finished on November 9, 2017

1

Question

```
6.1. Using the Wikipedia collection provided at the book website, create
a sample of stem clusters by the following process:
    1. Index the collection without stemming.
    2. Identify the first 1,000 words (in alphabetical order) in the index.
    3. Create stem classes by stemming these 1,000 words and recording
    which words become the same stem.
    4. Compute association measures (Dice?s coefficient) between all
    pairs of stems in each stem class. Compute co-occurrence at the
    document level.
    5. Create stem clusters by thresholding the association measure.
    All terms that are still connected to each other form the clusters.

Compare the stem clusters to the stem classes in terms of size and the
quality (in your opinion) of the groupings.
```

Answer

For this problem as well as other problems that used Galago it should be noted that I used Galago version 3.12 [4]. I also used the wiki-small html files as the input for the Galago search engine.

To handle the first two parts of this problem I decided to take advantage of the Galago search engine for indexing and sorting the index created by it. I then used galago to dump the unique keys from its inverted index using the following command:

$ galago dump-keys index/postings > dump-keys.csv

Galago creates an inverted index and it should be noted that from the keys dumped there were sequences of different characters but not all of them were actually words but rather numbers or misspellings. To compensate for this I used a python library called Pyenchant [5] to go through each and see if the the words were actually english and spelled correctly. The code to complete this is in **spellcheck.py** shown in Listing 1. This however still had numbers inside of it. To get to the english words starting with the letter 'a' I skipped to line 8578 and took the next 1000 words saved this to a file named **top-1k-words.txt**.

```
import enchant
import csv


def spellcheck():
    with open('./data/dump-keys.csv', 'r') as f, \
            open('./data/spell-checked-keys.csv', 'w') as out:
        reader = csv.reader(f)
        writer = csv.writer(out)
        d = enchant.Dict("en_US")
        for row in reader:
            word = row[0]
            if d.check(word):
                writer.writerow([word])


if __name__ == "__main__":
    spellcheck()
```

Listing 1: Spellcheck code in python

To stem each of the words in my list I used the NLTK python library [3] and specifically the PorterStemmer function to stem each word. I made a dictionary where each stem was a key and the stems then had an array of words that fell into that stem class and saved this to a json file named **stems.json**. A small sample of this is shown in Figure 1. I then used Galago again to find the document ids where the words had been used using the following Galago command:

$ galago dump-index index/postings > dump.csv

I then merged the previous dictionary with the document ids found in this step so I could use it to find the co-occurrence and Dice's Coefficient as described below. The code to perform these steps is in the file named **stemmer.py**.

Dice's Coefficient is described in our textbook [2] as:

$$\frac{n_{ab}}{n_a + n_b}$$

To compute Dice's Coefficient I used source code for the algorithm found en.wikibooks.org [] and used the python version. A subset of bigrams and their pair results of this are shown in Figure **??**, saved in **dice.csv**. The first column is the stem, the two words following are the comparisons, and the last value is the Dice Coefficient. The code used to create this is shown in Listing **??**, written in diceCluster.py.

```
import json
import csv


def dice_coefficient(a, b):
    """
    Taken directly from
    https://en.wikibooks.org/wiki/Algorithm_Implementation/
        Strings/Dice%27s_coefficient#Python
    """
    if not len(a) or not len(b):
        return 0.0
    """ quick case for true duplicates """
    if a == b:
        return 1.0
    """
    if a != b, and a or b are single chars, then
    they can't possibly match
    """
    if len(a) == 1 or len(b) == 1:
        return 0.0

    """ use python list comprehension, preferred over list.
        append() """
    a_bigram_list = [a[i:i + 2] for i in range(len(a) - 1)]
    b_bigram_list = [b[i:i + 2] for i in range(len(b) - 1)]

    a_bigram_list.sort()
    b_bigram_list.sort()

    # assignments to save function calls
    lena = len(a_bigram_list)
    lenb = len(b_bigram_list)
    # initialize match counters
    matches = i = j = 0
    while (i < lena and j < lenb):
        if a_bigram_list[i] == b_bigram_list[j]:
            matches += 2
            i += 1
            j += 1
        elif a_bigram_list[i] < b_bigram_list[j]:
            i += 1
        else:
            j += 1

    score = float(matches) / float(lena + lenb)
    return score

```

```
47
48 def dicePairs():
49     with open('data/stem-words-doc-ids.json') as f, \
50             open('data/dice.csv', 'w') as out:
51         data = json.load(f)
52         writer = csv.writer(out)
53         for stem, words in data.items():
54             # check bigrams of all stem words
55             for word in words:
56                 for w2 in words:
57                     if word != w2:
58                         dc = dice_coefficient(word, w2)
59                         writer.writerow([stem, word, w2, dc])
60
61
62 if __name__ == "__main__":
63     dicePairs()
```

Listing 2: Code to compute Dice coefficient and clusters directory

Figure 1: Subset of stems generated from Porter Stemmer

```
(venv) Grants-MBP:src gatkins$ cat data/dice.csv | head -n 20
abandon,abandoned,abandoning,0.7058823529411765
abandon,abandoned,abandonment,0.6666666666666666
abandon,abandoning,abandoned,0.7058823529411765
abandon,abandoning,abandonment,0.631578947368421
abandon,abandonment,abandoned,0.6666666666666666
abandon,abandonment,abandoning,0.631578947368421
abbrevi,abbreviated,abbreviation,0.7619047619047619
abbrevi,abbreviated,abbreviations,0.7272727272727273
abbrevi,abbreviation,abbreviated,0.7619047619047619
abbrevi,abbreviation,abbreviations,0.9565217391304348
abbrevi,abbreviations,abbreviated,0.7272727272727273
abbrevi,abbreviations,abbreviation,0.9565217391304348
abdic,abdicate,abdicating,0.75
abdic,abdicate,abdication,0.75
abdic,abdicating,abdicate,0.75
abdic,abdicating,abdication,0.7777777777777778
abdic,abdication,abdicate,0.75
abdic,abdication,abdicating,0.7777777777777778
abduct,abducted,abduction,0.6666666666666666
abduct,abducted,abductions,0.625
```

Figure 2: Subset of stems, the bigram pairs, and the dice coefficients for each

2

Question

`6.4. Assuming you had a gazetteer of place names available, sketch out an algorithm for detecting place names or locations in queries. Show examples of the types of queries where your algorithm would succeed and where it would fail.`

Answer

When thinking of an algorithm for detecting places names or locations in queries two very simple solutions come to mind right away. One is to tokenize the entire query and then search for each of the tokens directly in the index. This solution is of course a can be time consuming especially if the index is very large. Users never want to wait a long period of time because this method offers a slow solution.

Another simple solution is to have a cache of popular locations. Whenever a query goes out it is again tokenized and then the terms can be searched inside a small cache to save time. This however also has its own caveats of not covering every possible solution and actually increases time if it hits the cache first and doesn't find the location, but then also has to hit the index.

Another possible solution is to have predefined queries with modular variables in them. Some of the possible queries and their interchangeable variables are:

- "I went to Vegas this Summer." -> "I went to X this Summer."
- "Washington and Virginia are almost the same." -> "X and Y are almost the same."

These kinds of interchangeable queries could be stored procedures in a sense. This approach could also be used for bigrams since these are based on positions. This is a decent approach but there are many different ways to reference a location in a sentence therefore we can't contain cache of all the options. If we conform other queries to these predefined queries it can introduce more false positives. With this approach we also have to consider locations that aren't unigrams, such as "North Carolina."

An example algorithm is to combine each of these concepts into a single algorithm which updates overtime depending on the queries incoming. The goal is to have an algorithm that can receive the advantages of these methods

to increase query coverage, however there still might be locations that slip through each of these methods. My proposition is below in Listing 2.

Listing 3: Pseudo code algorithm for Location detection in queries

```
// perform lookup for term - increment cache frequency
    regardless
function cache_lookup(query):
        if query in query_cache:
                query_cache[query] += 1
                return true
        else
                query_cache[query] += 1
                return false

// check predefined query templates - return list of locations
function check_templates(query):
        locations = []
        for temp in templates:
                if query like template:
                        positions = template.location_positions
                        for pos in positions:
                                locations.append(query[pos])
                        return locations
        return locations

// long lookup to index
function long_lookup(tokens):
        locations = []
        for t in tokens:
                if find_token_index(t):
                        locations.append(t)
        return locations

// function to execute by default
function find_locations(query):
        if cache_lookup(query):
                return location(s)
        else
                locations = check_templates(query)
                if length(locs) != 0:
                        query_cache[query] += 1
                        return locations
                else
                        tokens = tokenize(query)
                        locations = long_lookup(tokens)
                        if length(locations) != 0:
                                return locations
        // no location found at all
```

```
        return error_no_locations

find_locations(query);
```

This pseudo code go through each of the above stated methods and checks. The order in which each executes should be the least costliest for each. If no locations are found then it will return 0 or error message of "no locations found." This algorithm also has an update policy to the cache for frequent queries.

3

Question

Answer

4

Question

Answer

5

Question

Answer

References

[1] Atkins, Grant. "CS734 Assignment 3 Repository" Github. N.p., 9 November 2017. Web. 9 November 2017.https://github.com/grantat/cs834-f17/tree/master/assignments/A3.

[2] B. Croft, D. Metzler, and T. Strohman. "Search Engines: Information Retrieval in Practice." Pearson, 2009. Web. 14 October 2017. ISBN 9780136072249.

[3] Bird, Steven, Edward Loper and Ewan Klein (2009). "Natural Language Processing with Python" O?Reilly Media Inc. Web. 9 November 2017.http://www.nltk.org/.

[4] "The Lemur Project - Galago". Web. 9 November 2017. https://sourceforge.net/p/lemur/wiki/Galago/.

[5] "Pyenchant". Web. 9 November 2017. http://pythonhosted.org/pyenchant/.