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 [6] and used the python version. A subset of bigrams and their pair results of this are shown in Figure 2, 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 2, written in diceCluster.py.

Dice clusters were then calculated using:

$$\frac{2 * doc\_count_{ab}}{doc\_count_a + doc\_count_b}$$

The threshold I set was 0.1 and I got a relatively high number of clusters, 100 as shown with a subset show in Figure 3. The difference is noticeable when comparing stem classes to stem clusters. The influence of document frequency shows that this changes the cluster associations greatly.

```
1  import json
2  import csv
3
4
5  def dice_coefficient(a, b):
6      """
7      Taken directly from
8      https://en.wikibooks.org/wiki/Algorithm_Implementation/
           Strings/Dice%27s_coefficient#Python
9      """
10     if not len(a) or not len(b):
11         return 0.0
12     """ quick case for true duplicates """
13     if a == b:
14         return 1.0
15     """
16     if a != b, and a or b are single chars, then
17     they can't possibly match
18     """
19     if len(a) == 1 or len(b) == 1:
20         return 0.0
21
22     """ use python list comprehension, preferred over list.
           append() """
23     a_bigram_list = [a[i:i + 2] for i in range(len(a) - 1)]
24     b_bigram_list = [b[i:i + 2] for i in range(len(b) - 1)]
25
26     a_bigram_list.sort()
27     b_bigram_list.sort()
28
29     # assignments to save function calls
30     lena = len(a_bigram_list)
31     lenb = len(b_bigram_list)
32     # initialize match counters
33     matches = i = j = 0
34     while (i < lena and j < lenb):
```

```
35          if a_bigram_list[i] == b_bigram_list[j]:
36              matches += 2
37              i += 1
38              j += 1
39          elif a_bigram_list[i] < b_bigram_list[j]:
40              i += 1
41          else:
42              j += 1
43
44      score = float(matches) / float(lena + lenb)
45      return score
46
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              if len(words) > 1:
55                  # check bigrams of all stem words
56                  temp = set()
57                  for word in words:
58                      for w2 in words:
59                          if word != w2 and (word, w2) not in temp
                                and \
60                                  (w2, word) not in temp:
61                              dc = dice_coefficient(word, w2)
62                              temp.add((word, w2))
63                              writer.writerow([stem, word, w2, dc
                                    ])
64                  # print(temp)
65
66
67  def docs(search_word, stems):
68      for stem, words in stems.items():
69          for word, vals in words.items():
70              if word == search_word:
71                  return vals["doc_ids"]
72
73
74  def diceCluster(wa, wb, stems):
75      # document association
76      # print(wa, wb)
77      docs_wa = docs(wa, stems)
78      wa_len = len(docs_wa)
79
80      docs_wb = docs(wb, stems)
81      wb_len = len(docs_wb)
```

```
82
83      wab_docs = set(docs_wa).intersection(docs_wb)
84      wab_len = len(wab_docs)
85      score = (2 * wab_len) / (wa_len + wb_len)
86      return score
87
88
89  if __name__ == "__main__":
90      dicePairs()
91      threshold = 0.1
92      with open('data/stem-words-doc-ids.json') as f, \
93              open('data/dice.csv') as f2, \
94              open('data/dice-clusters.csv', 'w') as out:
95          stems = json.load(f)
96          reader = csv.reader(f2)
97          writer = csv.writer(out)
98          for row in reader:
99              wa = row[1]
100             wb = row[2]
101             dc = diceCluster(wa, wb, stems)
102             if dc > threshold:
103                 writer.writerow([row[0], wa, wb, dc])
```

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,abandonment,0.631578947368421
abbrevi,abbreviated,abbreviation,0.7619047619047619
abbrevi,abbreviated,abbreviations,0.7272727272727273
abbrevi,abbreviation,abbreviations,0.9565217391304348
abdic,abdicate,abdicating,0.75
abdic,abdicate,abdication,0.75
abdic,abdicating,abdication,0.7777777777777778
abduct,abducted,abduction,0.6666666666666666
abduct,abducted,abductions,0.625
abduct,abduction,abductions,0.9411764705882353
aberr,aberration,aberrations,0.9473684210526315
abid,abide,abiding,0.6
abil,abilities,ability,0.7142857142857143
abl,able,ables,0.8571428571428571
ablut,ablution,ablutions,0.9333333333333333
abnorm,abnormal,abnormalities,0.7368421052631579
abnorm,abnormal,abnormality,0.8235294117647058
abnorm,abnormal,abnormally,0.875
```

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

```
(venv) Grants-MBP:src gatkins$ cat data/dice-clusters.csv | head -n 20
abduct,abduction,abductions,0.10526315789473684
aberr,aberration,aberrations,0.2
abil,abilities,ability,0.13766730401529637
abomin,abomination,abominations,0.14285714285714285
abridg,abridged,abridges,0.125
abridg,abridged,abridgment,0.11764705882352941
abridg,abridging,abridgment,0.4
abrog,abrogated,abrogation,0.25
absolv,absolve,absolved,0.125
abut,abuts,abutting,0.3333333333333333
academ,academic,academics,0.13504823151125403
acceler,accelerate,accelerations,0.2
acceler,accelerates,accelerations,0.26666666666666666
acceler,accelerating,accelerator,0.11764705882352941
acceler,accelerator,accelerators,0.16
acclam,acclamation,acclamations,0.3333333333333333
accomplic,accomplice,accomplices,0.16666666666666666
account,accountancy,accountants,0.16
account,accountant,accountants,0.17647058823529413
accredit,accreditation,accredited,0.14457831325301204
```

Figure 3: 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 3.

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

6.5. Describe the snippet generation algorithm in Galago. Would this algorithm work well for pages with little text content? Describe in detail how you would modify the algorithm to improve it.

Answer

The snippet generation algorithm in Galago is actually publicly available and can be found inside the source code in the following directory: galago-3.12/core/src/main/java/org/lemurproject/galago/core/index/corpus/SnippetGenerator.java. Although the code is slightly overbearing, and hideous java, the algorithm to create snippets algorithm is pretty straightforward. A short description is as follows:

1. First hits a "getSnippet" function which takes document texts and a set of query terms
2. It then finds the positions of the query terms in the document and passes it to the next functionality
3. It then tokenizes and stems each word in the query using the Krovetz Stemmer
4. For each of the terms matched in the documents it creates a region, or sentence of variable length shown in the document.
5. If the terms in the snippet have regions that overlap it attempts to join them, if the size of the snippet is greater than 150 characters it will attempt to reduce the region size
6. It then adds separators between regions by "..." but also bolds the words if found verbatim.

This algorithm does not work well with little text content. This algorithm focuses on query matches in the text, but also extracts regions for each of these snippets. If the region size isn't large enough the snippet might think the context is not enough for a snippet. An example is shown in Figure 4. The first image shows the SERP page for "bit" where the top

two results have good snippets with this word in context. The third entry however, has no snippet and in Figure 5 the document doesn't actually contain the entire query word but actually is a stem for the word "bits". I think this could be better represented with simply creating regions around the words with "bit" stem if there are no exact matches of "bit" in context.

I think I would improve the algorithm with inclusion of stemming logic in a snippet if there are no entries. If there aren't even any other stemmed words in the document then there should be some kind of abstract text to better represent the document. There are signs in the code that there are actually attempts to find the stems of the words and expose words create weights for the "best candidate" snippets. In the notes its stated that this actually is too time consuming and will force users to wait for results. So in this case they chose speed over appearances in some cases which is a fair choice.

Figure 4: SERP page for “bit” query

From Wikipedia, the free encyclopedia

Ladder-DES

General	
Designers	Terry Ritter
First published	February 22, 1994
Derived from	DES
Related to	DEAL
Cipher detail	
Key sizes	224 bits
Block sizes	128 bits
Structure	Nested Feistel network
Rounds	4
Best public cryptanalysis	
Eli Biham's attacks require 2^{36} plaintext-ciphertext pairs	

In cryptography, **Ladder-DES** is a block cipher designed in 1994 by Terry Ritter. It is a 4-round Feistel cipher with a block size of 128 bits, using DES as the round function. It has no actua
size is 4×56=224 bits.

In 1997, Eli Biham found two forms of cryptanalysis for Ladder-DES that depend on the birthday paradox; the key is deduced from the presence or absence of *collisions*, plaintexts that give
encryption process. He presented both a chosen-plaintext attack and a known-plaintext attack; each uses about 2^{36} plaintexts and 2^{90} work, but the known-plaintext attack requires much mo

Figure 5: Actual Document for bit entry

4

Question

7.7. What is the ‘‘bucket’’ analogy for a bigram language model? Give examples.

Answer

In the bigram language model the “bucket” analogy is very similar to the analogy as unigrams as well. In the unigram model the analogy of the “bucket” is the idea of reaching into an bucket of words for a document, reading off a word, and then putting it back in the bucket. Its probabilistic that words with a high frequency or are important will be drawn more often than others.

The bigram model is very similar except for the fact that it retrieves two word pairs from a bucket which represents a document. The two word pairs are words that are adjacent to one another in a document, therefore providing ordering of words. These bigrams also provide more context than a unigram model since they provide this ordering. For example, the sentence “North Carolina citizens wants to outlast the storm” would be create the following list in a unigram language model:

- “North”
- “Carolina”
- “citizens”
- “wants”
- “outlast”
- “storm”

This has all of the stop word removed but these words become representative of the document that are in. In the bigram language model the following list would be created from the previous list:

- “North Carolina”
- “Carolina citizens”
- “citizens wants”

- "wants outlast"
- "outlast storm"

If a user picks out the word "North" from the unigram bucket it proves to be a very ambiguous word. However, if we pick out the first bigram which contains "North" it becomes "North Carolina." This brings a lot more context as well shows connections between words. The frequency of the these word pairs will prove to be a better representative of the document than that of the unigrams. This also shows that the second word of the bigrams are dependent on the first word. As the occurrences of bigrams starting with "North" increase whichever second word for the bigram starts to become more representative for the distribution. So if there are two "North Carolina" occurrences its apparent that this has more weight over one "North Virginia" occurrence to represent a document.

5

Question

7.8. Using the Galago implementation of query likelihood, study the impact of short queries and long queries on effectiveness. Do the parameter settings make a difference?

Answer

After rummaging through the code for Galago 3.12 and reading the Galago wiki [4] I believe Galago uses a Dirichlet algorithm for query likelihood. One of the direct downfalls to this implementation is that if a query contains a term that is not present inside of a document it will consider the documents to have no relevance to the query regardless of the other terms in the query. The code for the Dirichlet scoring is show in Listing 4.

Listing 4: Scoring Algorithm in DirichletScorer.java

```
public double score(int count, int length) {
  double numerator = count + (mu * background);
  double denominator = length + mu;
  return Math.log(numerator / denominator);
}
```

When comparing the impact of short queries versus long queries its better to say that the average query length performs the best. This is due to the fact that they contain enough terms to form a topic model to represent the query. If we took a query with one word, then there could be multiple topics with which represent that word and the topic that has the highest probability for that word would be favored highly when it might not be the topic we were hoping for. For long queries its not that different. If a query contains many words then there could be noise in the selection of the best topic. If all words have a high probability for a different topic it will be difficult to take into account who to select for the best result.

I don't think the parameter settings would make a drastic difference unless made to create a drastic difference. Parameters won't have much influence on the size of a query input by a user. One of the ways it could help is that it could help match queries to the correct topics, however it shouldn't change the ordering of the results on a SERP.

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/.

[6] "Wikibook - Dice's Coefficient". Web. 9 November 2017. https://en.wikibooks.org/wiki/Algorithm_Implementation/Strings/Dice%27s_coefficient#Python.