Assignment 3

Fall 2016
CS834 Introduction to Information Retrieval
Dr. Michael Nelson

Mathew Chaney

November 6, 2016

Contents

List of Figures

Listings

List of Tables

1 Question 6.1

1.1 Question

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.

1.2 Approach

The `stem.py` script, found in Listing 3, was used to solve this problem.

2 Question 6.2

2.1 Question

Create a simple spelling corrector based on the noisy channel model. Use a single-word language model, and an error model where all errors with the same edit distance have the same probability. Only consider edit distances of 1 or 2. Implement your own edit distance calculator (example code can easily be found on the Web)

2.2 Approach

The `spelling.py` script, found in Listing 2, was created using the Python programming language [1]. Downloaded `big.txt` to calculate an example language model. These words were counted and stored in a map that was compressed on disk using the pickle python library [2].

The process of determining a spelling correction is as follows:

1. Count all the words contained in the example text file. This count is used to determine the language model $P(W)$ calculation.
2. Take the input word and determine all existing (correctly spelled) words with edit distance one and two.
3. With the assumption that shorter edit distances equate to a higher probability of being the correct spelling, select from the remaining set of (valid) words the one with the shortest edit distance and highest value for $P(W)$.

$P(W)$ is calculated with the following formula:

$$P(W) = \frac{C_W}{N}$$

where C_W is the word count for word W and N is the sum of all word counts.

2.3 Results

Here is some sample output from the spelling.py script.

```
[mchaney@mchaney-l spelling]$ ./spelling.py werds
words
[mchaney@mchaney-l spelling]$ ./spelling.py flewerz
flower
[mchaney@mchaney-l spelling]$ ./spelling.py splling
selling
[mchaney@mchaney-l spelling]$ ./spelling.py sweling
swelling
[mchaney@mchaney-l spelling]$ ./spelling.py aacck
back
```

Listing 1: spelling.py example output

3 Question 6.3

3.1 Question

Implement a simple pseudo-relevance feedback algorithm for the Galago search engine. Provide examples of the query expansions that your algorithm does, and summarize the problems and successes of your approach.

3.2 Approach

Here's a formula.

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

4 Question 6.5

4.1 Question

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.

4.2 Answer

Snippet creation is done by the `SnippetGenerator` class. This class takes as parameters to it's `getSnippet` method the document text as a `String` and a `Set` of `String` query terms, and returns a `String` that is a query-relevant snippet, or summary, of the document.

The snippet generator begins by turning the document text into a list of tokens for processing. The generator then parses these tokens, looking for query term matches, and when it finds a match, it creates a `SnippetRegion` object that stores the location within the document where the query term matched, plus five contextual terms preceding and following each term match. This equates to storing sentence fragments containing query terms.

After collecting all of the regions in the document containing a query term the generator begins constructing the final snippet by adding the SnippetRegions found from the previous step, combining those regions that overlap each other into larger regions, until a final list of `SnippetRegions` is created with total length in terms is no greater than 40 + the length of the last `SnippetRegion` added.

With the final list of `SnippetRegions` the algorithm builds an HTML string containing all the snippets concatenated together for rendering the snippet in a browser while adding `<strong>` tags around each query term match for emphasis.

This approach favors regions at the beginning of the document without regard to query context. One way to improve upon this method is to count the number of query terms found in the initial overlapped regions selected for the final snippet by assigning a score to each region based on the frequency of query term matches found within that region. This could potentially cut down the size of the final snippet by choosing regions that contain more query words within the normal extent of 5 terms per query word match.

5 Appendix

```
#!/usr/bin/env python

import re
import sys
import cPickle
from collections import Counter


def get_words():
    try:
        return cPickle.load(open('words.p', 'rb'))
    except IOError:
        wordmap = Counter(re.findall(r'\w+', open('big.txt').read().lower()))
        cPickle.dump(wordmap, open('words.p', 'wb'))
        return cPickle.load(open('words.p', 'rb'))


words = get_words()
N = sum(words.values())


def exists(wordset):
    return set([word for word in wordset if word in words])


def prob(word):
    return float(words[word]) / float(N)


def edit1(w):
    letters    = 'abcdefghijklmnopqrstuvwxyz'
    deletes    = [w[:i]+w[i+1:]               for i in range(len(w))]
    transposes = [w[:i]+w[i+1]+w[i]+w[i+2:] for i in range(len(w)-1)]
    replaces   = [w[:i]+l+w[i+1:]             for i in range(len(w)) for l in letters]
    inserts    = [w[:i]+l+w[i:]               for i in range(len(w)+1) for l in letters]
    return set(deletes + transposes + replaces + inserts)


def edit2(word):
    e2 = [edit1(w) for w in edit1(word)]
    return [item for sublist in e2 for item in sublist]


def parse(word):
    return exists([word]) or exists(edit1(word)) or exists(edit2(word)) or [word]


def correct(word):
    return max(parse(word), key=prob)


if __name__ == '__main__':
    print correct(sys.argv[1])
```

Listing 2: spelling.py

```
1 #!/usr/bin/env python
```

Listing 3: stem.py

6 References

[1] The Python Programming Language. Available at: https://www.python.org/. Accessed: 2016/09/17.

[2] Python.org. Python object serialization. Available at: https://docs.python.org/2/library/pickle.html. Accessed: 2016/11/06.