CS-734/834 Introduction to Information Retrieval: Assignment #3: Ex 6.1, 6.2, 6.5, MLN1 & MLN2

Due on Thursday, November 10, 2016

Dr. Michael L. Nelson

Plinio Vargas
pvargas@cs.odu.edu

Contents

List of Figures

Listings

List of Tables

1 Exercise 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.

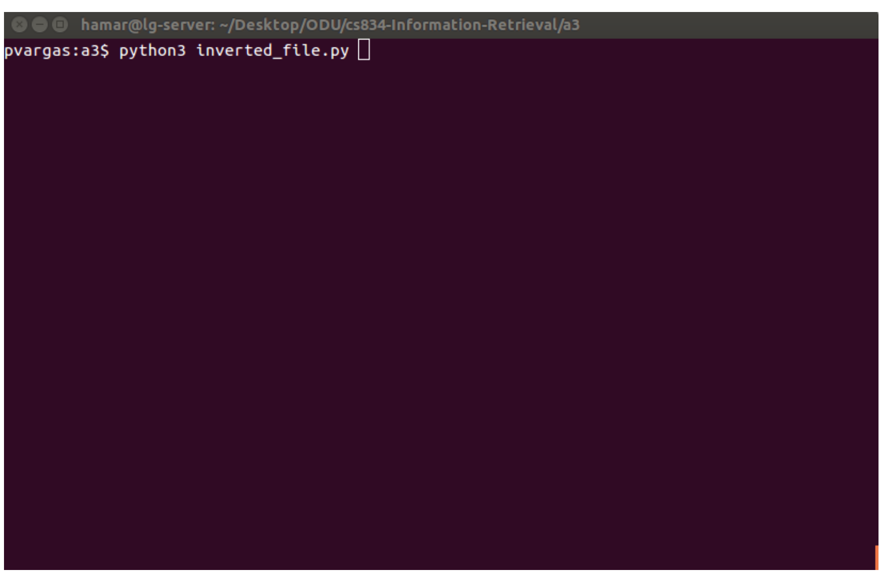
Compare the stem clusters to the stem classes in terms of size and the quality (in your opinion) of the groupings.

1.1 Approach

For step (1), we used the unigram generated from the previous assignment. The word collection is under the file name *zift_law.txt*. We developed a python script **inverted_file.py** to index the collection. The collection can be indexed by typing from the command prompt:

\# **python3 inverted_file.py**

Figure 1: Indexing Collection



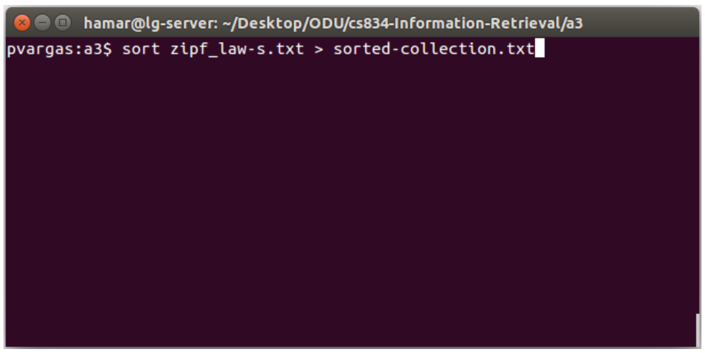
Exercise 6.1 continued on next page...

A document root where the collection is located is passed to generate a file (*file-path.txt*) containing all resources in the collection. The file will be used later on as a document index for the collection. The unigram index was sorted by typing unix command:

sort zipf_la-s.txt > sorted-collection.txt

The new file **sorted-collection.txt** vocabulary is indexed in alphabetical order. The sorted-collection file was uploaded as a list object in memory, and passed as a parameter to generate the index (see Listing 1 line 30).

Figure 2: Sort Unigram



The inverted file index will be generated using format **999:1**, where **999** points to the document location and **1** is the term frequency at the document level (lines 64-68).

Listing 1: inverted_file.py

```
16 __date__ = 'Mon ,November 7 at 7:49:11'
17 __email__ = 'pvargas@cs.odu.edu'
18 
19 file_no = 0
20 word_index = {}
21 file_path = []
22 
23 
24 def create_index(path, term_file):
25     vocabulary = []
26     with open(term_file, 'r') as f:
27         for word in f:
28             vocabulary.append(word.strip())
29 
30     get_url(path, vocabulary)
31 
32     # store inverted file
33     with open('inverted-file.txt', 'w') as f:
34         for index in word_index:
35             print(index, word_index[index])
36             f.write('%s:%s\n' % (index, word_index[index]))
37 
38     # store file path
39     with open('file-path.txt', 'w') as f:
```

Exercise 6.1 continued on next page...

```python
        for path in file_path:
            f.write('%s\n' % path)

    return


def get_url(url, vocabulary):
    global file_no, word_index, file_path
    if os.path.isfile(url):
        file_no += 1
        print(file_no, url)
        file_path.append(url)

        # get file content
        f = open(url, 'r')
        page = f.read()
        f.close()

        soup = BeautifulSoup(page, 'html.parser')
        data = soup.body.get_text()
        data = re.sub('[*#/=?&>}{!<)(;,|\"\.\[\]]', ' ', data.lower())

        counts = Counter(data.split())

        for word in vocabulary:
            if word in counts:
                word_index.setdefault(word, [])
                word_index[word].append('%d:%d' % (file_no, counts[word]))
                print('%s %d:%d' % (word, file_no, counts[word]), end=',')
        print()
        del data, page, soup

    if not os.path.isdir(url):
        return

    for filename in os.listdir(url):
        get_url(os.path.join(url, filename), vocabulary)

    return

if __name__ == '__main__':
    # record running time
    start = time()
    print('Starting Time: %s' % strftime("%a,  %b %d, %Y at %H:%M:%S", localtime()
        ))

    create_index('./en', 'vocabulary.txt')

    print('\nEnd Time:  %s' % strftime("%a,  %b %d, %Y at %H:%M:%S", localtime()))
    print('Execution Time: %.2f seconds' % (time()-start))
    sys.exit(0)
```

To identify the first 1,000 words (in alphabetical order), we developed **stemmer.py**. Its main purpose is to filter non-English words from the collection.

Listing 2: stemmer.py

```
__author__ = 'Plinio H. Vargas'
__date__ = 'Sat,November 5 at 23:11:30'
__email__ = 'pvargas@cs.odu.edu'


def get_top_words(filename, n):
    dictionary = PyDictionary()
    k = 0
    line_no = 0
    first_n = []
    is_english = ''
    with open(filename, 'r') as f:
        for line in f:
            line_no += 1
            word = line.split()[0]
            if len(line.split()) > 1:
                qty = int(line.split()[1])
            else:
                qty = 0
            print(word, line_no)
            try:
                print(word, word[0], is_english, k)
                if qty > 1 and word[0] >= 'a':
                    is_english = dictionary.meaning(word)
                    if is_english:
                        first_n.append(word)
                        k += 1
                        if k > n:
                            break
            except IndexError:
                print("Yes")
                pass

            print(k)
    print(first_n)

    with open('vocabulary.txt', 'w') as f:
        for word in first_n:
            f.write('%s\n' % word)

    f.close()

    return


if __name__ == '__main__':
    # checks for argument

    # record running time
    start = time()
    print('Starting Time: %s' % strftime("%a,  %b %d, %Y at %H:%M:%S", localtime()
```

```
        ))

    get_top_words("sorted-collection.txt", 1000)

    print('\nEnd Time:  %s' % strftime("%a,  %b %d, %Y at %H:%M:%S", localtime()))
    print('Execution Time: %.2f seconds' % (time()-start))
    sys.exit(0)
```

Stemmer.py uses the **PyDictionary** library to find if a word is in the English dictionary. It iterates through a loop until the first 1,000 English words are verified.

For step(4) we developed **cluster.py** to generate the Dice's coefficient between all pairs of stems. We imported the library *Stemmer* to stem our 1,000 word collection. The 1,000 words were loaded in memory (see Listing 3).

Listing 3: cluster.py

```
__author__ = 'Plinio H. Vargas'
__date__ = 'Sat ,November 5 at 23:11:30'
__email__ = 'pvargas@cs.odu.edu'

window = 100
inverted_index = {}
file_path = []


def get_top_words(filename):
    stemmer = Stemmer.Stemmer('english')
    cluster = {}

    # stem and cluster n-top words in sorted collection
    with open(filename, 'r') as f:
        for word in f:
            word = word.strip()
            stem = stemmer.stemWord(word)
            cluster.setdefault(stem, [])
            cluster[stem].append(word)

    # calculate dice's coefficient for cluster pairs
    for values in sorted(cluster.items(), key=operator.itemgetter(1)):
        values = values[0]

        # add cluster header
        f = open('cluster-association.txt', 'a')

        print()
        f.write('\n')
        print(values, cluster[values])
        f.write('%s - %s\n' % (values, cluster[values]))
        f.close()

        if len(cluster[values]) > 1:
            # get number of pairs in cluster
```

Exercise 6.1 continued on next page...

```
            n = len(cluster[values]) - 1
            n = int(n * (n + 1) / 2)

            # initialize pair frequency
            window_freq = [0 for x in range(n)]

            dice_coefficient(cluster[values], window_freq)

    return


def dice_coefficient(pairs, window_freq):
    global window

    # add cluster header
    f = open('cluster-association.txt', 'a')

    # calculate pair coefficient
    for i in range(len(pairs)):
        for k in range(i + 1, len(pairs)):
            print('\t', pairs[i], pairs[k], end=' -- dice-coef:')
            f.write(('\t (%s, %s) -- dice-coef:' % (pairs[i], pairs[k])))

            # inverted_index[pairs[i]] contains documents for first pair element
            pair1 = [x.split(':') for x in inverted_index[pairs[i]]]
            pair2 = [x.split(':') for x in inverted_index[pairs[k]]]

            # find files where term intersect
            no_intercept = 0
            for files in pair1:
                if files[0] in [x[0] for x in pair2]:
                    no_intercept += 1
                    url = file_path[int(files[0])]

                    """
                    f = open(url, 'r')
                    page = f.read()
                    f.close()

                    soup = BeautifulSoup(page, 'html.parser')
                    data = soup.body.get_text()
                    data = re.sub('[*#/=?&>}{!<)(;,|\"\.\[\]]', ' ', data)

                    del data, page, soup
                    """

            print('%.4f' % (2 * no_intercept / (len(pair1) + len(pair2))))
            f.write(('%.4f\n' % (2 * no_intercept / (len(pair1) + len(pair2)))))
    f.close()
    return
```

The script generates a file, clustering words with equal stem. To calculate the coefficient between pairs we generate:

$$\sum_{i=1}^{n-1} i$$

iterations, since we have a double loop from $i \rightarrow n$ outside $i+1 \rightarrow n$. See Listing 3 lines 70-101. Inside this double loop, we pair all possible combinations of words clustered with similar stem and calculate the Dice's coefficient by considering the documents where the pair intercept. Since we have the information from the index file, we can extract the term frequency from the document and make our calculation:

$$\frac{\mathrm{n}_{ab}}{\mathrm{n}_a \cdot \mathrm{n}_b}$$

A cluster sample with Dice's coefficient is shown below:

1.2 Solution

/air, aire, aired, aires, airing, airings, airs

Table 1: Cluster by Stem: air

Pair	Dice's Coef
(air, aire)	0.0116
(air, aired)	0.1022
(air, aires)	0.0166
(air, airing)	0.0399
(air, airings)	0.0118
(air, airs)	0.0676
(aire, aired)	0.0000
(aire, aires)	0.0000
(aire, airing)	0.0000
(aire, airings)	0.0000
(aire, airs)	0.0000
(aired, aires)	0.0190
(aired, airing)	0.1702
(aired, airings)	0.0488
(aired, airs)	0.1633
(aires, airing)	0.0000
(aires, airings)	0.0000
(aires, airs)	0.0000
(airing, airings)	0.1818
(airing, airs)	0.1579
(airings, airs)	0.0769

Using a threshold of 0.1500 we can re-cluster to:

/air, aired
/aired, airing, airs

/airing, airings, airs

2 Exercise 6.2

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).

2.1 Approach

Our approach is based on the probability distribution model $P(w)$. The probability of a given word will be obtained from the small wikipedia collection and term frequency already generated on the previous assignment. The file *zipf_law-s.txt* was used to make the probability comparison among words.

A list object was used to upload the dictionary in memory (lines 98-100 Listing 4). Four misspelled words were placed in an array to test our algorithm (line 26).

Since the entire collection consists of over 240,000 words before the distance calculation is performed, we only considered words in which length had a delta of 2 or less (lines 37-40). We passed the misspelled word and filtered dictionary to get the distance calculation (line 44).

In order to reduce the list size we placed each string of the list in a set to filter the words with similar number of letters. For example, if the misspelled word is *teh*, the set for this word will be {'e', 'h', 't'} a comparison with the correct word *the* will result with the same set {'e', 'h', 't'}. We do a subtraction between the misspelled word and the terms in the filter array. Only the words with a distance less or equal to the max distance allowed will be considered (lines 66-72).

Using this approach does not guarantee the words are going to be considered in equal sequence. In other words, when making a set comparison, the order of the strings are irrelevant. The accomplished task was to ensure the number of characters between two words are within our threshold. Finally, to calculate the distance, we count the number of characters that have similar sequence with the misspelled word. An array containing a very small list is returned to calculate their probabilities (lines 74-90).

Finally, if $P(w/w_p) > P(e/w)$ then we will select w. From the remaining short list, we compare all their probabilities and select the word with the highest probability (lines 46-56).

Listing 4: spell-checker.py

```
24 def main():
25     max_distance = 2
26     non_words = ['Teh', 'couse', 'tremmor', 'stodent']
27 
28     # upload collection
29     dictionary = []
30     with open('zipf_law.txt', 'r', encoding='iso-8859-1') as f:
31         for word in f:
32             dictionary.append(word.strip().split('\t'))
```

```

    for term in non_words:
        w_length = len(term)
        possible_list = []
        for word in dictionary:
            distance = abs(w_length - len(word[0]))
            if distance <= max_distance:
                possible_list.append(word[0].lower())

        collection_size = len(dictionary)

        short_list = distance_calc(term.lower(), possible_list)

        highest_prob = 0
        idx = 0
        for word in short_list:
            prob_w = float(dictionary[possible_list.index(word[0])][1]) / collection_size

            if highest_prob < prob_w:
                if abs(word[1] - len(term)) < max_distance:
                    highest_prob = prob_w
                    idx = possible_list.index(word[0])

        print('For %s the correct spelling is --> %s' % (term, possible_list[idx]))

    return


def distance_calc(word, possible_list):
    word_set = set([x for x in word])

    new_list = []
    distance = 2
    for p_word in possible_list:
        p_word_set = set([x for x in p_word])
        new_distance = len(p_word_set-word_set)

        if new_distance <= distance:
            new_list.append(p_word)
            distance = new_distance

    short_list = []
    max_count = 0
    for p_word in new_list:
        correct_count = 0
        i = 0
        for char in p_word:
            for k in range(i,len(word)):
                if char == word[k]:
                    correct_count += 1
                    i = k + 1
```

```
                        break

            if max_count < correct_count:
                max_count = correct_count
                short_list.append((p_word, max_count))

    print(short_list)

    return short_list
```

2.2 Solution

Executing the program produced the following results:

```
Starting Time: Thu,  Nov 10, 2016 at 22:26:46
[('the', 2), ('teeth', 3)]
For Teh the correct spelling is --> the
[('the', 1), ('user', 3), ('ouse', 4)]
For couse the correct spelling is --> ouse
[('recent', 2), ('terms', 3), ('report', 4), ('tremor', 6)]
For tremmor the correct spelling is --> tremor
[('under', 1), ('contents', 3), ('street', 4), ('students', 6)]
For stodent the correct spelling is --> students

End Time:  Thu,  Nov 10, 2016 at 22:26:48
Execution Time: 2.00 seconds

Process finished with exit code 0
```

3 Exercise 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.

3.1 Background

The Galago snippet generation algorithm is related to the concept of relevance feedback. If a user makes a query, it will be important to know if the document we are about to retrieve is relevant to that query. So, the snippet must be a summary of the document in relation to our query.

Galago snippet algorithm derives from the work Luhn in the 1950s. Luhn explored the concept of ranking and selecting the top sentences in a document using a *significant factor*. In order to accomplish this, we have to find which are the significant words in the document. He used the concept of word frequency as an indicator of how significant the words were for the document.

Exercise 6.5 continued on next page...

```
w  w  w  w  w  w  w  w  w  w  w.
```
(Initial sentence)

```
w  w  s  w  s  s  w  w  s  w  w.
```
(Identify significant words)

```
w  w [s  w  s  s  w  w  s] w  w.
```
(Text span bracketed by significant words)

3.2 Algorithm

The code for generating **galago** query snippets was found in the **Lemur Project** at `https://sourceforge.net/p/lemur/galago/ci/c15406935ce7e697d7a8d0ce329606f100276921/tree/core/src/main/java/org/lemurproject/galago/core/index/corpus/SnippetGenerator.java#l11`

The code was written in Java. Below are some portions of the code:

```java
// Goals:  1. find as many terms as possible
//         2. find terms that are close together
//         3. break on sentences when possible (?)
// BUGBUG: might not have all the terms highlighted here
public ArrayList<SnippetRegion> combineRegions(final ArrayList<SnippetRegion> regions) {
    ArrayList<SnippetRegion> finalRegions = new ArrayList();
    SnippetRegion last = null;
    int snippetSize = 0;
    int maxSize = 40;

    for (int i = 0; i < regions.size(); i++) {
        SnippetRegion current = regions.get(i);

        if (last == null) {
            last = current;
        } else if (last.overlap(current)) {
            SnippetRegion bigger = last.merge(current);

            if (bigger.size() + snippetSize > maxSize) {
                finalRegions.add(last);
                last = null;
            } else {
                last = bigger;
            }
        } else if (last.size() + snippetSize > maxSize) {
            break;
        } else {
            finalRegions.add(last);
            snippetSize += last.size();
            last = current;
        }
```

```
        }

        if (last != null && snippetSize + last.size() < maxSize) {
            finalRegions.add(last);
        }

        return finalRegions;
    }
```

Then, in summary the algorithm:

1. Find as many terms as possible
2. Find terms that are close together
3. Break on sentences when possible

Would this algorithm work well for pages with little text content?

This algorithm will not work well with little text context because the ranking of sentences will not provide enough information to make a good comparison on what region of the document is more significant than others. Since the base of the algorithm is ranking regions or sentences, then all sentences are going to become equal and without distinction among each other.

Describe in detail how you would modify the algorithm to improve it

Since the algorithm works well with plenty text:

1. We need to define a threshold where the amount of text is not enough.
2. Stem our query terms to find other possible terms related in the text.
3. Expand our query terms to find other possible terms related in the text.
4. Use the stems and expansion results to generate new sentences ranking.
5. Identify regions where the added terms are more significant.
6. Extract and display the result.

4 Exercise MLN1

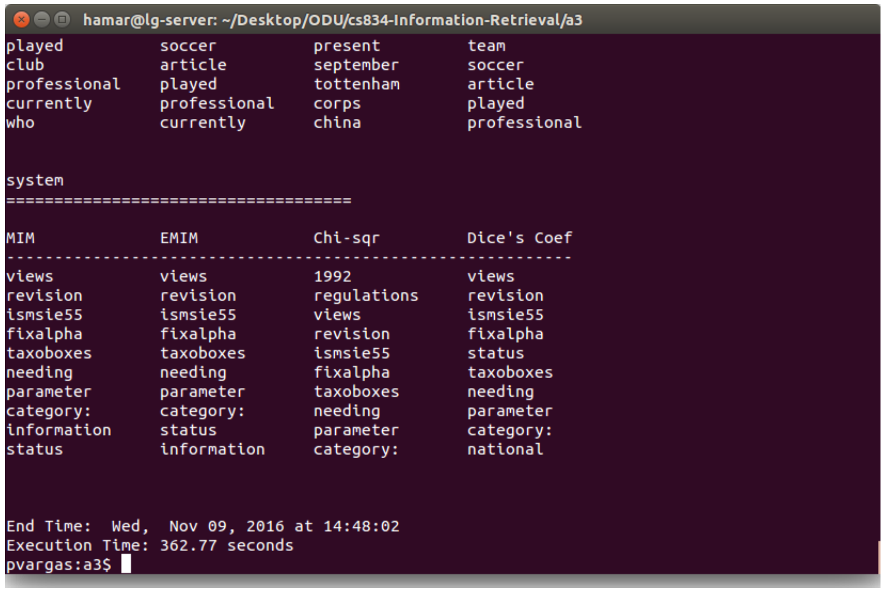
MLN1: using the small wikipedia example, choose 10 words and create stem classes as per the algorithm on pp. 191-192

4.1 Approach

We modified problem 6.1 to consider only 10 words instead of 1000. So the approach and implementation are the same.

4.2 Solution

Figure 3: Stem Class



5 Exercise MLN2

Using the small wikipedia example, choose 10 words and compute MIM, EMIM, chi square, dice association measures for full document & 5 word windows (cf. pp. 203-205)

5.1 Approach

Words selected:

/altarpiece, resurrection, retirement, football, country, system, book, california, department, washington
The problem was divided into three sub-problems. Each sub-problem was related to a module developed within python script **term-association.py**.

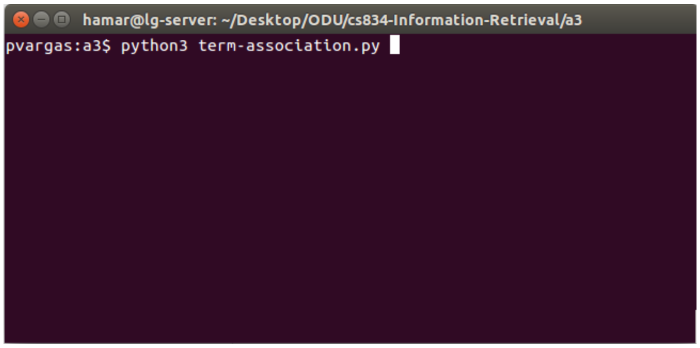
- Getting file index.
- Getting file content.
- Obtaining term frequency.

5.1.1 Running Script

We can generate the term association by typing from the command prompt:

 # python3 term-association.py

Figure 4: Generating Term Association



5.1.2 Main Module

This module directs the script to accomplish various tasks, and provide the required data prior to its execution. Since we did not remove the stop words from the index, they have to be identified and removed prior to the term association calculation, otherwise we will be certainly associating stop-words with our terms, thus providing undesired results. These stop-words in Listing 5 line 23, were obtained by inspecting our collection term frequency in file:*zipf_law.txt*. The higher frequency terms were visually inspected, copied and plugged into the list object **stop-words**. This object was used for comparison and removal of those existing words in the body of inspected documents. There were some non-stop-words added to this object that did not meet the frequency requirement to be considered stop-words, but were added because they became part of the generated index, but they were meaningless to our term association.

The list of terms for which are we going to find the association in the collection are loaded in a list object (**vocabulary**) in line 32. The inverted file index is loaded into memory, and as it was stated before, it has the form *term* [999:*freq*,], where 999 refers to the document index in *inverted-file.txt* and *freq* is the frequency of the term at the document level.

The list of terms contained in **vocabulary** is inspected to ensured they are contained in our inverted file index. In case the term is not included, it will be inspected and added to the inverted file object for the collection (lines 50-52).

The window size (line 58) and the terms to be associated (**vocabulary**) are passed as a parameter (lines 58-59) to find and measure all the terms within the window proximity of the terms. An object (**term_data**) is returned containing the frequency of words associated to **vocabulary** at the document level. This information will be used (line 61) to calculate the association using various measures: *MIM, EMIM, Chi-square* and *Dice's coefficient*.

Listing 5: term-association.py - main module

```
20 inverted_index = {}
21 file_path = []
22 stop_words = ['the', 'of', 'and', 'a', 'in', 'to', 'wikipedia', 'is', 'by', 'was',
       'for', 'on', 'from', 'edit', 'this',
23               'as', 'with', '1', 'about', 'user', '3', 'it', 'page', 'he', 'free',
                     'that', 'at', 'registered', '2',
24               'all', 'his', 'help', 'if', 'an', 'see', '^', 'c', 'under', 'u', '
```

```
                  window', 'contents', 'or', 'are',
              '2008', 'also', 'be', 's', '4', '5', '6', 'v', 'i', '0-86124-352-8',
                   'articlediscussioncurrent'
             ]


def main():
    # terms to calculate
    vocabulary = ['altarpiece', 'resurrection', 'retirement', 'football', 'country
        ', 'system', 'book', 'california',
                  'department', 'washington']
    index_2add = []

    # get all urls
    with open('file-path.txt', 'r') as f:
        for url in f:
            file_path.append(url.strip())

    # get inverted index
    #with open('test.txt', 'r') as f:
    with open('inverted-file.txt', 'r') as f:
        for line in f:
            r = re.search('(^.*?):(.*)', line.strip())
            inverted_index[r.group(1)] = re.sub("[,\[\]\']", ' ', r.group(2)).
                split()

    print('Size of index:', len(inverted_index))

    for word in vocabulary:
        if word not in inverted_index:
            index_2add.append(word)

    build_index(index_2add)

    print('Size of index:', len(inverted_index))

    window = 5
    term_data = get_term_freq(vocabulary, window)

    assoc_measure = calc_assoc_measure(term_data)

    for term in assoc_measure:
        print(term,'\n===================================\n')
        print('{0:15} {1:15} {2:15} {3:15}'.format('MIM', 'EMIM', 'Chi-sqr', 'Dice
            \'s Coef'))
        print('-------------------------------------------------------------')
        r1 = sorted(assoc_measure[term], key=lambda l:l[1], reverse=True)
        r2 = sorted(assoc_measure[term], key=lambda l:l[2], reverse=True)
        r3 = sorted(assoc_measure[term], key=lambda l:l[3], reverse=True)
        r4 = sorted(assoc_measure[term], key=lambda l:l[4], reverse=True)

        mim = []
        for row in r1[:10]:
```

```
73              mim.append(row[0])
74
75          emim = []
76          for row in r2[:10]:
77              emim.append(row[0])
78
79          chi = []
80          for row in r3[:10]:
81              chi.append(row[0])
82
83          dice = []
84          for row in r4[:10]:
85              dice.append(row[0])
86
87          for i in range(10):
88              print('{0:15} {1:15} {2:15} {3:15}'.format(mim[i], emim[i], chi[i],
                    dice[i]))
89
90          print('\n')
91
92      return
```

Finally, the values provided from the different measures are sorted in descending order (line 63-91) to display the $k = 10$ words strongly associated with a particular methodology.

5.1.3 Getting file index

This module generates the index of any term in the parameter list that was not included in the inverted file. The parameter list is an array of terms by which its association measure will be calculated (**vocabulary**). An iteration is performed on all the resources in our collection to inspect its content. The document index is stored in **file_no** (line 132) and increased by one every time a new document gets inspected.

The document contents are extracted from module *get_file_content()* (line 135) and the frequency of all the terms within the document are obtained using the python library **Counter** (line 138).

Listing 6: term-association.py - build index module

```
128     global inverted_index, file_path
129     file_no = 0
130
131     for url in file_path:
132         file_no += 1
133
134         # get file content
135         data = get_file_content(url)
136
137         # get term frequency within document
138         counts = Counter(data)
139
140         # include term document frequency into index
141         for word in vocabulary:
142             if word in counts:
143                 inverted_index.setdefault(word, [])
144                 inverted_index[word].append('%d:%d' % (file_no, counts[word]))
```

```
145
146     return
```

Finally, if a term in **vocabulary** is found inside the document, then it is added to the index using format *term* [999:*freq*]. See lines 141-144.

5.1.4 Getting file content

Since this functionally is commonly used to solve our main problem, in order to maintain consistency and repeated work, a module was created to extract the content of a particular web-page resource. The module takes as a parameter the location where the resource is stored, reads the file and removes all the contents of no interest such as [,= etc. The return value is an array containing all terms within the document.

Listing 7: term-association.py - get content module

```
def get_file_content(url):
    # open file to get raw content
    f = open(url, 'r')
    page = f.read()
    f.close()

    soup = BeautifulSoup(page, 'html.parser')
    data = soup.body.get_text()
    data = re.sub('[*#/=?&>}{!<)(;,|\"\.\[\]]', ' ', data.lower())

    del page, soup

    return data.split()
```

5.1.5 Obtaining terms frequencies

The meat of our solution resides within this module. The main scope is to pair a document with a term or list of terms. Then, for each term, find the location (index) where the term is positioned within the document. By inspecting a window size w to the left and right of the current position, we can record the words with proximity w to our term. The last step is to find out the frequency of those words within proximity w to our term within the document. The process is repeated for all the documents in the collection. The resulting object is a dictionary of dictionaries containing a term and their words with proximity w to their frequency:

$$\{\textit{term}\text{: } \{\textit{word1:freq}, \textit{word2:freq}, \cdots\ \}\}$$

The list of terms (**vocabulary**) for which we are going to measure their association and the *window* size is passed as parameter to this module. The inverted file provides the resources where these terms are located. For every document where a term is found in the collection (Listing 8 line 154), we can find the frequency of the term by splitting data value [999:*freq*] and obtaining the left side of the colon (':') or the pointer to the resource (line 155).

The content of the file is stored in the list object **data** (line 157), then all the stop-words are remove from **data** (lines 160-166). The frequency of words in **data** can be found using library *Counter*, the result is stored into dictionary object **counts** (line 168).

Then, we proceed to find the first position where the *term* is located within **data** object (line 171). First, we get *window* number of words to the left of our current position (lines 175-179) taking in consideration that a number of words could point n positions before the beginning of **data** array. Then, we do the same going to the right of our current *term* position taking in consideration that a number of words could point n positions beyond the end of **data** array (lines 181-187).

To make a distinction between the frequency that a word appears within the document and the frequency a word appears within a *window* with a *term* in the document the '_' character was added to the dictionary to indicate this distinction. This frequency calculation is performed in lines 189-203 Listing 8.

Since, it is possible that a *term* could be found more than once with in a document we have to find the next position where the *term* is located in **data** object. We will continue this process until no more *terms* are found in the object (lines 206-232). Finally, the result is a dictionary of dictionary (**window_term**) in line 234.

Listing 8: term-association.py - Get Term Frequency Module

```
149 def get_term_freq(vocabulary, window):
150     window_term = {}
151     for term in vocabulary:
152         window_term[term] = {}
153         # get term frequency per document
154         for file_index in inverted_index[term]:
155             ptr = int(file_index.split(':')[0]) - 1
156             # get file content
157             data = get_file_content(file_path[ptr])
158             print(ptr, file_path[ptr])
159
160             # remove stop-words
161             for value in stop_words:
162                 while value in data:
163                     try:
164                         data.remove(value)
165                     except ValueError:
166                         break
167
168             counts = Counter(data)
169
170             # get window terms
171             pos = data.index(term)
172             n = len(data)
173
174             # get terms left-side of window
175             left = pos - window
176             if left < 0:
177                 left = 0
178
179             left_window = data[left:pos]
180
181             # get terms right-side of window
182             right = pos + window + 1
183
```

Exercise MLN2 continued on next page...

```
            if right > n:
                right = n

            right_window = data[pos + 1:right]

            for value in right_window:
                if '_' + value not in window_term[term]:
                    window_term[term]['_' + value] = 1
                if value not in window_term[term]:
                    window_term[term][value] = counts[value]
                else:
                    window_term[term]['_' + value] += 1

            for value in left_window:
                if '_' + value not in window_term[term]:
                    window_term[term]['_' + value] = 1
                if value not in window_term[term]:
                    window_term[term][value] = counts[value]
                else:
                    window_term[term]['_' + value] += 1

            cycle = True
            while cycle or right < n:
                try:
                    pos += data[right + 1:].index(term) + window + 2

                    left = pos - window
                    left_window = data[left:pos]
                    for value in left_window:
                        if '_' + value not in window_term[term]:
                            window_term[term]['_' + value] = 1
                        if value not in window_term[term]:
                            window_term[term][value] = counts[value]

                    right = pos + window + 1
                    if right > n:
                        right = n

                    right_window = data[pos + 1:right]

                    for value in right_window:
                        if '_' + value not in window_term[term]:
                            window_term[term]['_' + value] = 1
                        if value not in window_term[term]:
                            window_term[term][value] = counts[value]

                except ValueError:
                    cycle = False
                    break

    return window_term
```

5.2 Solution

Table 2: Association Measure for Word “Altarpiece”

MIM	$EMIM$	X^2	$Dice$
works	works	madonna	works
cupboard-shaped	cupboard-shaped	church	cupboard-shaped
size	size	portrait	size
onesti	onesti	child	onesti
baronci	baronci	clock	baronci
placed	placed	st	placed
choir	choir	astronomical	choir
bardi	bardi	virgin	bardi
retablo	retablo	knight	retablo
one	one	gothic	one

Most strongly associated words for “altarpiece” in small wikipedia collection.

Table 3: Association Measure for Word ”Resurrection”

MIM	$EMIM$	X^2	$Dice$
works	works	madonna	works
cupboard-shaped	cupboard-shaped	church	cupboard-shaped
size	size	portrait	size
onesti	onesti	child	onesti
baronci	baronci	clock	baronci
placed	placed	st	placed
choir	choir	astronomical	choir
bardi	bardi	virgin	bardi
retablo	retablo	knight	retablo
one	one	gothic	one

Most strongly associated words for “resurrection” in small wikipedia collection.

Table 4: Association Measure for Word “Retirement”

MIM	$EMIM$	X^2	$Dice$
after	after	1992	until
until	until	regulations	after
which	which	amendment	which
following	following	health	references
before	references	service	following
but	before	national	time
cdata	but	scotland	before
references	cdata	after	life
former	former	until	1989
announced	announced	smith	career

Most strongly associated words for “resurrection” in small wikipedia collection.

Table 5: Association Measure for Word “Football”

MIM	$EMIM$	X^2	$Dice$
encyclopedia	club	1992	club
national	encyclopedia	order	player
team	national	amendment	league
soccer	team	brazil	encyclopedia
article	league	scotland	national
played	soccer	present	team
club	article	september	soccer
professional	played	tottenham	article
currently	professional	corps	played
biographical	currently	china	professional

Most strongly associated words for “football” in small wikipedia collection.

Table 6: Association Measure for Word “Country”

MIM	$EMIM$	X^2	$Dice$
language	united	state	united
english	state	language	states
state	states	united	state
administration	language	1992	language
encyclopedia	region	english	region
province	english	states	county
time	administration	administration	location
united	county	region	english
genre	location	encyclopedia	administration
running	encyclopedia	province	encyclopedia

Most strongly associated words for “country” in small wikipedia collection.

Table 7: Association Measure for Word "System"

MIM	$EMIM$	X^2	$Dice$
revision	revision	1992	revision
views	views	regulations	views
ismsie55	ismsie55	revision	ismsie55
fixalpha	fixalpha	views	fixalpha
parameter	parameter	ismsie55	status
taxoboxes	taxoboxes	fixalpha	parameter
needing	needing	parameter	taxoboxes
category:	category:	taxoboxes	needing
information	status	needing	category:
status	information	category:	national

Most strongly associated words for "system" in small wikipedia collection.

Table 8: Association Measure for Word "Book"

MIM	$EMIM$	X^2	$Dice$
published	published	published	published
comic	comic	comic	comic
references	first	symmetry	first
isbn	references	brazil	series
her	isbn	nfl	which
first	her	02	book
who	who	samples	references
wrote	wrote	06	new
encyclopedia	new	wollstonecraft	isbn
life	encyclopedia	ufo	her

Most strongly associated words for "book" in small wikipedia collection.

Table 9: Association Measure for Word "California"

MIM	$EMIM$	X^2	$Dice$
san	university	caries	university
state	san	brazil	san
university	state	san	los
school	school	nfl	angeles
encyclopedia	los	dental	southern
born	angeles	university	state
santa	southern	her	school
southern	encyclopedia	darling	states
first	born	coach	california
age	santa	hosted	encyclopedia

Most strongly associated words for "california" in small wikipedia collection.

Table 10: Association Measure for Word "Department"

MIM	$EMIM$	X^2	$Dice$
france	france	france	france
article	article	article	article
stub	stub	stub	communes
département	département	département	stub
geographical	geographical	geographical	département
you	you	you	geographical
canton	canton	canton	you
administration	administration	administration	region
country	country	country	canton
arrondissement	arrondissement	arrondissement	administration

Most strongly associated words for "department" in small wikipedia collection.

Table 11: Association Measure for Word "Washington"

MIM	$EMIM$	X^2	$Dice$
post	post	brazil	post
west	west	lead	dc
school	dc	fox	union
categories	school	nfl	university
wayne	union	news	george
township	categories	games	west
seattle	university	detroit	new
dc	wayne	jets	county
south	township	texas	state
encyclopedia	seattle	iraq	school

Most strongly associated words for "washington" in small wikipedia collection.

Exercise MLN2 continued on next page...

The association measure seems to work remarkable well. If we take a look at the terms associated with the word 'washington' on Table 11 we can see that using the Mutual Information measure (*MIM*), **Washington** is associated with the "Washington Post", "West Washington", "Washington school", "Washington DC". The *EMIM* measure is very similar to the *MIM*, but in different ranking order.

The Chi-square for the same term seems to be more related with sports words related to the word **Washington**, such as fox, nfl, games, jets, etc. There is also a good association with the word **Washington** using the *Dice's coefficient*