CS-734/834 Introduction to Information Retrieval:
Assignment #1:
Ex 1.1, 3.7, 3.8, 3.9, 3.14

Due on Thursday, September 22, 2016

Dr. Michael L. Nelson

Plinio Vargas
pvargas@cs.odu.edu

Contents

List of Figures

Listings

1 Exercise 1.1

Think up and write down a small number of queries for a web search engine. Make sure that the queries vary in length (i.e., they are not all one word). Try to specify exactly what information you are looking for in some of the queries. Run these queries on two commercial web search engines and compare the top 10 results for each query by doing relevance judgments. Write a report that answers at least the following questions: What is the precision of the results? What is the overlap between the results for the two search engines? Is one search engine clearly better than the other? If so, by how much? How do short queries perform compared to long queries?

1.1 Approach

I was interested in finding out the recipe of a Latin-American dish called **Maduro**. **Google** and **Bing** were the two search engines used to compare the relevance of the top 10 webpages returned. The experiment started by typing two keywords: **Maduro plate**.

For our purpose, we will use the top 10 results as the total number of retrieved documents. To compare the precision between the two searches, we can use the formula:

$$precision = \frac{|\{relevant\ documents\} \cap \{retrieved\ documents\}|}{|\{retrieved\ documents\}|} \tag{1}$$

1.1.1 2-Word Query

Google Results:
The top four queries returned were related to a cigar brand and Venezuelan President: Nicolas Maduro. There were three sites ranked 5^{th}, 6^{th} and 8^{th}, related to Maduro as a dish, but did not included the recipe. See Figure 1

Bing Results:
The 1^{st} and 7^{th} were in fact Maduro dish recipes. The 3^{rd}, 6^{th} and 9^{th} were related to a Maduro dish, but they did not provide a recipe. The remaining sites were related to the Maduro cigar brand. See Figure 2

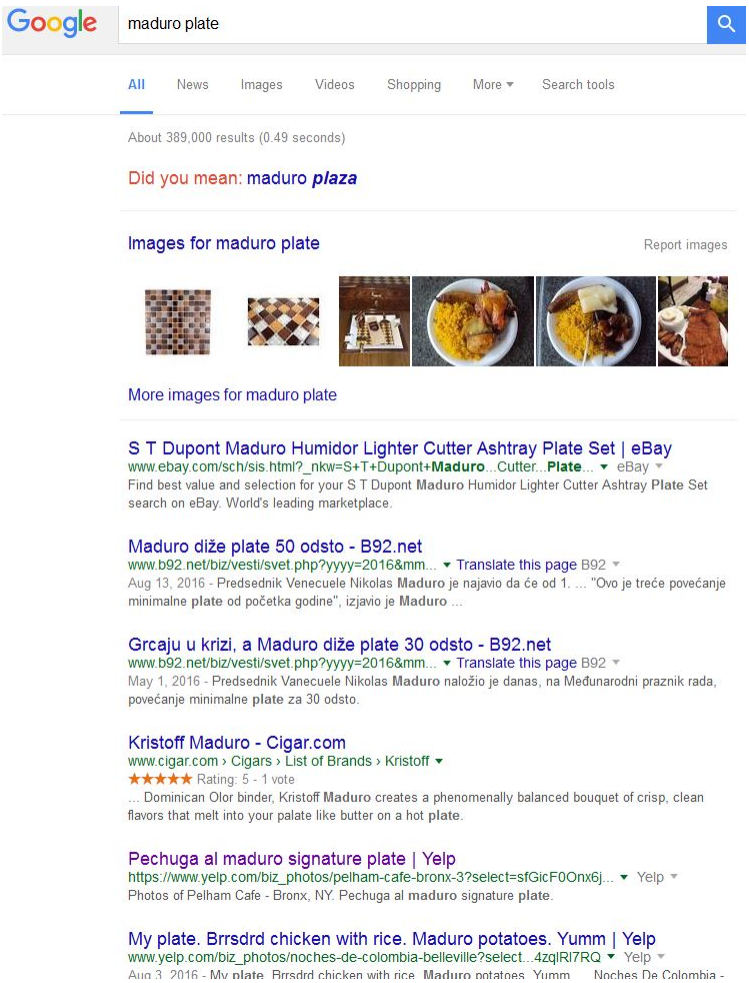
First query precision calculation for keyword: **Maduro plate**

$$Google\ precision = \frac{|\{\ \} \cap \{1 \cdots 10\}|}{|10|} = 0$$

$$Bing\ precision = \frac{|\{1, 7\} \cap \{1 \cdots 10\}|}{|10|} = \frac{1}{5}$$

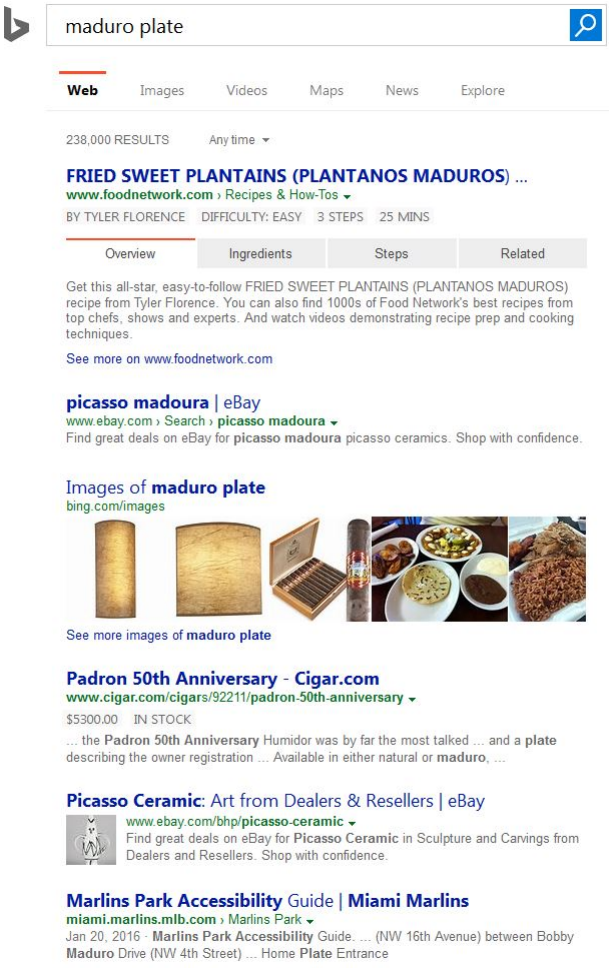
Exercise 1.1 continued on next page...

Figure 1: Google Keyword Search **Maudro Plate**



Exercise 1.1 continued on next page...

Figure 2: Bing Keyword Search **Maudro Plate**



Exercise 1.1 continued on next page...

1.1.2 3-Word Query

For the second query, three keywords were typed: **Maduro plantain plate**
Google Results:
The top result returned was a Wiki page containing plantain information. The remaining 9 results included Maduro recipe information.

Bing Results:
The 1st, 2nd, 4th, 5th, 6th, 7th, 8th and 10th were in fact Maduro dish recipes. The remaining sites were related to Maduros, but did not contain a recipe.

Precision calculation for the 3 keyword query:

$$Google\ precision = \frac{|\{\ 2 \cdots 10\} \cap \{1 \cdots 10\}|}{|10|} = \frac{9}{10}$$
$$Bing\ precision = \frac{|\{1, 2, 4, 5, 6, 7, 8, 10\} \cap \{1 \cdots 10\}|}{|10|} = \frac{4}{5}$$

Although Google precision was better than Bing, we have also to consider the top ranked result in Google was not relevant, while Bing's top rank result was relevant.

Surprisingly, these two search engines did not overlap in their result, rather they complemented each other.

1.1.3 4-Word Query

For the third query, four keywords were typed: **Maduro plantain plate recipe**

Google Results:
All the results contained recipe information.

Bing Results:
The 1st, 2nd, 4th, 5th, 6th, 7th, 8th and 10th were in fact Maduro dish recipes. The 3rd result was related to **Maduro Recipe**, however it was a page pointing to other pages containing the recipe.

Precision calculation for the 4-keyword query:

$$Google\ precision = \frac{|\{\ 1 \cdots 10\} \cap \{1 \cdots 10\}|}{|10|} = \frac{10}{10}$$
$$Bing\ precision = \frac{|\{1, 2, 4, 5, 6, 7, 8, 9, 10\} \cap \{1 \cdots 10\}|}{|10|} = \frac{9}{10}$$

Exercise 1.1 continued on next page...

1.2 Solution

Table 1: Google vs Bing Comparison

Number Key Words Precision	Search Engine	
	Google	Bing
2-words	0.0000	0.2000
3-words	0.9000	0.8000
4-words	1.0000	0.9000
Average	0.6333	0.6333

Therefore, if we do a comparison strictly by precision numbers, we have a tie. However, we can appreciate that as the number of keywords increases, the precision for **Google** gets better in comparison with **Bing**.

2 Exercise 3.7

Write a program that can create a valid sitemap based on the contents of a directory on your computer's hard disk. Assume that the files are accessible from a website at the URL http://www.example.com. For instance, if there is a file in your directory called homework.pdf, this would be available at http://www.example.com/homework.pdf. Use the real modification date on the file as the last modified time in the sitemap, and to help estimate the change frequency.

2.1 Approach

Python script *sitemap.py*, shown on Listing **??**, was developed to complete this exercise. This problem can be divided into the following sub-problems:

1. File discovery: Given a path, find all files within the path (including directories).
2. Tag modification date: Obtains the modification date attribute from all files in a given path.
3. Frequency estimation: Estimates change frequency using modification date as a reference.
4. Formatting: Format data using sitemap XML structure.

2.1.1 File Discovery

Script *sitemap.py* uses recursively **depth** first traversal to discover all files in a given path. The path is assigned on line 19. The list of files are obtained in lines 24-25. They are inspected in the function **sitemap** (lines 33-58). The condition to finish the recursion is given on lines 53-54. If the file is not a directory, then the recursion ends. Otherwise it will continue until it reaches the bottom of the branch.

2.1.2 Tag Modification Date

We can find the last modified attribute to the file, using the *Python* library **os.stat**. See line 33 on Listing **??**.

2.1.3 Frequency Estimation

To estimate the frequency of the crawl, *sitemap.py* compares the system date with last modified file attribute (lines 34-44).

2.1.4 Formatting

Script *sitemap.py* uses the convention provided on [?] for output formatting. Tags are wrapped around at each level when an object's attribute gets discovered. See lines 22-23, 26, 48-52.

Listing 1: sitemap.py

```
13
14 def main():
15     # record running time
16     start = time()
17     print('Starting Time: %s\n' % strftime("%a,  %b %d, %Y at %H:%M:%S", localtime
           ()))
18
19     path = '.\\documents'
20     level = 1
21
22     print('<?xml version="1.0" encoding="UTF-8"?>')
23     print('<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">')
24     for filename in os.listdir(path):
25         sitemap(os.path.join(path, filename), level)
26     print('</urlset>')
27
28     print('\nEnd Time:  %s' % strftime("%a,  %b %d, %Y at %H:%M:%S", localtime()))
29     print('Execution Time: %.2f seconds' % (time()-start))
30     return
31
32
33 def sitemap(url, level):
34     epoc_time = os.stat(url).st_mtime
35     now = datetime.today()
36     previous = datetime.strptime(strftime('%Y-%m-%d', localtime(epoc_time)), "%Y-%
           m-%d")
37     no_days = (now - previous).days
38
39     if no_days < 3:
40         freq = 'daily'
41     elif no_days < 32:
42         freq = 'weekly'
43     else:
44         freq = 'monthly'
45
46     sitmap_url = url.replace('\\', '/').strip('..')
47     if os.path.isfile(url):
48         print('<url>')
49         print('  <loc>http://www.example.com{0}</loc>\n'.format(sitmap_url), end='
               ')
50         print('  <lastmod>{0}</lastmod>'.format(strftime('%Y-%m-%d', localtime(
               epoc_time))))
51         print('  <changefreq>%s</changefreq>' % freq)
```

Exercise 3.7 continued on next page...

```
52          print('</url>')
53      if not os.path.isdir(url):
54          return
55
56      for filename in os.listdir(url):
57          sitemap(os.path.join(url, filename), level + 1)
58      return
```

2.2 Solution

Sitemap from local computer under the class assignment folder named ./document.

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
<url>
  <loc>http://www.example.com/documents/cs834_pvargas_hw1.aux</loc>
  <lastmod>2016-09-20</lastmod>
  <changefreq>daily</changefreq>
</url>
<url>
  <loc>http://www.example.com/documents/cs834_pvargas_hw1.lof</loc>
  <lastmod>2016-09-20</lastmod>
  <changefreq>daily</changefreq>
</url>
<url>
  <loc>http://www.example.com/documents/cs834_pvargas_hw1.log</loc>
  <lastmod>2016-09-20</lastmod>
  <changefreq>daily</changefreq>
</url>
<url>
  <loc>http://www.example.com/documents/cs834_pvargas_hw1.lol</loc>
  <lastmod>2016-09-20</lastmod>
  <changefreq>daily</changefreq>
</url>
<url>
  <loc>http://www.example.com/documents/cs834_pvargas_hw1.out</loc>
  <lastmod>2016-09-20</lastmod>
  <changefreq>daily</changefreq>
</url>
<url>
  <loc>http://www.example.com/documents/cs834_pvargas_hw1.pdf</loc>
  <lastmod>2016-09-20</lastmod>
  <changefreq>daily</changefreq>
</url>
<url>
  <loc>http://www.example.com/documents/cs834_pvargas_hw1.synctex.gz</loc>
  <lastmod>2016-09-20</lastmod>
  <changefreq>daily</changefreq>
</url>
```

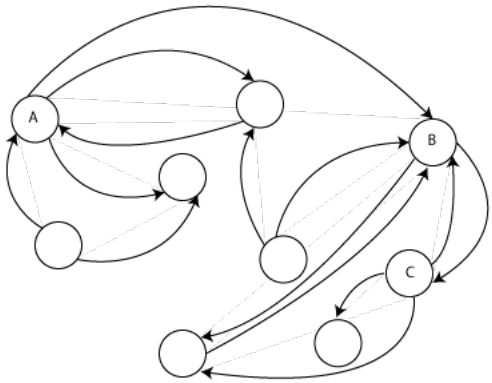
```
<url>
  <loc>http://www.example.com/documents/cs834_pvargas_hw1.tex</loc>
  <lastmod>2016-09-20</lastmod>
  <changefreq>daily</changefreq>
</url>
<url>
  <loc>http://www.example.com/documents/cs834_pvargas_hw1.toc</loc>
  <lastmod>2016-09-20</lastmod>
  <changefreq>daily</changefreq>
</url>
<url>
  <loc>http://www.example.com/documents/images/blogclust.jpg</loc>
  <lastmod>2016-09-19</lastmod>
  <changefreq>weekly</changefreq>
</url>
<url>
  <loc>http://www.example.com/documents/images/blogclustP5.jpg</loc>
  <lastmod>2016-09-19</lastmod>
  <changefreq>weekly</changefreq>
</url>
<url>
  <loc>http://www.example.com/documents/images/blogs2d.jpg</loc>
  <lastmod>2016-09-19</lastmod>
  <changefreq>weekly</changefreq>
</url>
<url>
  <loc>http://www.example.com/documents/images/Thumbs.db</loc>
  <lastmod>2016-09-19</lastmod>
  <changefreq>weekly</changefreq>
</url>
<url>
  <loc>http://www.example.com/documents/table-1.tex</loc>
  <lastmod>2016-09-19</lastmod>
  <changefreq>weekly</changefreq>
</url>
</urlset>
```

3 Exercise 3.8

Suppose that, in an effort to crawl web pages faster, you set up two crawling machines with different starting seed URLs. Is this an effective strategy for distributed crawling? Why or why not?

Yes, this is an effective strategy. If we consider that both machines have the same starting URLs,the probability of trying to reach to the same links will increase, since neighboring nodes most likely will have similar links inter-connected. Looking at figure **??** we can see that nodes B and C have a closer relationship to the adjacent nodes thank node A.
Then, the further apart they are (in terms of similarity) less is the change of pointing to the same link. In the case of our two machine having a different staring seed URL will decreasing the possibility of crawling the same link sooner than if they started at the same initial point, making the propose approach more effective.

Figure 3: Web-crawler



4 Exercise 3.9

Write a simple single-threaded web crawler. Starting from a single input URL (perhaps a professor's web page), the crawler should download a page and then wait at least five seconds before downloading the next page. Your program should find other pages to crawl by parsing link tags found in previously crawled documents.

4.1 Approach

Python script *crawler.py*, shown on Listing **??**, was developed to complete this exercise. This problem can be divided into the following sub-problems:

1. Extract Links: Given the representation of a given resource, all the links within that object are extracted for further analysis.

Exercise 3.9 continued on next page...

2. Validate Links: *crawler.py* trusts the information provided by the server. ONLY objects with a Content-type value of "text/html" and a 200 response code are accepted. Redirects (RC 301 or 302) are followed to the pointed location.

3. Track Crawled Links: This sub-problem is critical in the implementation to avoid an indefinite loop when two or more links point to each other. This was accomplished by keeping a hash list with URLs already crawled.

4.1.1 Extract Links

Crawler.py uses **Python** library *BeautifulSoup* to extract all the links from returned object. There are other details that *crawler.py* takes care off to optimize the extraction before going through the validation step, such as ensuring the URL ends with a backslash and that its schema is accurate. Lines 46-63 take care of link extraction in *crawler.py*.

4.1.2 Validate Links

The validation is done by inspecting the header returned by the Webserver. The first validation (line 70) ensures the representation is not for an image, pdf, text file, etc. It ONLY accepts HTML content. If the resource is available, the server will send a response code 200. Only then, that link is crawled (lines 71-72). There are times where the resources are moved to a different location pointed by the URL. If the server has this information, a status code 301 or 302 is sent back to the client requesting that page. *Crawler.py* validates this response 71-87. The iteration is performed until a valid location is reached (response 200) or a different response code is given by the server.

4.1.3 Track Crawled Links

It is very common for webpages in a lower hierarchy to point to the home page. It is also possible that pages in the same hierarchy point to each other. If one does not keep track of which pages have been crawled, not only will the process get duplicated, but it also could get into an indefinite loop, as previously stated. *Crawler.py* first encodes the URL (line 25), and finds out if this hash is in the hash table (line 26). The crawling is recursive, and the condition to end the recursion is when the webpage has already been crawled.

Listing 2: crawler.py

```
21 def crawler(url, crawled_pages):
22     url = url.strip()
23
24     # create hash for URI
25     encoded_url = md5(url.encode()).hexdigest()
26     if encoded_url in crawled_pages:
27         print('%s already crawled..' % url)
28         return
29
30     crawled_pages[encoded_url] = url
31
32     #locale.setlocale(locale.LC_ALL, 'en_US.utf8')
33     print('Extracting links from: %s\n' % url)
34
35     # get uri status
36     if requests.get(url).status_code != 200:
37         print('\n\nURI is not available from SERVER. Verify URI.\n')
38         return
39
```

```
    # get source from URI
    page = requests.get(url).text

    # get parse hostname from URI
    url = 'http://' + urlparse(url).netloc

    # create BeautifulSoup Object
    soup = BeautifulSoup(page, 'html.parser')

    # place source link into list
    for link in soup.find_all('a'):
        uri = link.get('href')
        try:
            # include hostname if url is provided by reference
            if ((len(uri) > 6 and uri[:7].lower() != 'http://') or len(uri) < 7)
                and uri[:8].lower() != 'https://':
                if uri[:2] == '//':    # if url has double backslash then url is
                    not provided by reference
                    uri = 'http:' + uri
                elif uri[0] != '/':    # include backslash if it was not include
                    by reference
                    uri = url + '/' + uri
                else:
                    uri = url + uri
        except TypeError:
            print('%s is invalid' % url)
            return

        # for debugging
        #print(uri)

        try:
            r = requests.head(uri)
            if 'Content-Type' in r.headers and r.headers['Content-Type'] == 'text/
                html':
                if r.status_code == 200:
                    crawler(uri, crawled_pages)
                elif 'location' in r.headers and (r.status_code == 301 or r.
                    status_code == 302):
                    counter = 1
                    while counter < 7:
                        try:
                            uri = r.headers['location']
                            r = requests.head(r.headers['location'])
                            if 'location' in r.headers and (r.status_code == 301
                                or r.status_code == 302):
                                counter += 1
                            elif r.status_code == 200:
                                crawler(uri, crawled_pages)
                            else:
                                break
                        except KeyError:
                            print('Couldn\'t find resource for: %s' % url)
```

```
87                              break
88
89          except requests.exceptions.SSLError:
90              print('Couldn\'t open: %s. URL requires authentication.' % uri)
91          except requests.exceptions.ConnectionError:
92              print('Couldn\'t open: %s. Connection refused.' % uri)
93
94      return
```

4.2 Solution

Our single thread *crawler* started on URL <http://cs.odu.edu/~mln/>. It spent quite an amount of time looping through the ODU domain. Eventually, it entered an outside domain. The process had to be interrupted since it entered the World Wide Web, and even though, the same link does not get crawled twice, there were many requests to crawl the same page.

5 Exercise 3.14

Write a program to generate simhash fingerprints for documents. You can use any reasonable hash function for the words. Use the program to detect duplicates on your home computer. Report on the accuracy of the detection. How does the detection accuracy vary with fingerprint size?

5.1 Approach

Python script *simhash.py*, shown on Listing **??**, was developed to complete this exercise. We used three files: data1.txt, data2.txt and data3.txt (uploaded in Github). They are read and pass as a reference to *simhash.py*. The first two data files were modified slightly to simulate a plagiarism, while the third file is completely different. We can divide this problem into the following sub-problems:

1. Remove Stop Words
2. Hash Document Words
3. Apply Simhash Algorithm according with [**?**]

5.1.1 Remove Stop Words

Using **Python** library *stop_words* gives us the ability to remove all the stops words from our file in one line (21).

5.1.2 Hash Document Words

simhash.py applies MD5 hash to all the words from the file (line 34). The hash is converted to a binary form (line 35). The preceding zeros in Python are removed, and replaced with a b. That is the reason for taking only the element beyond position 2 in the array (line 35).

5.1.3 Apply Simhash Algorithm

The *simhash* algorithm sums the frequency of the words with a value 1 in the hash, and deducts its frequency with a value zero (lines 33-43).

Listing 3: simhash.py

```
def simhash(data):
    # remove stop words from file
    clean_data = [x for x in data.split() if x not in stop_words]

    # initialize weight vector
    weight_vector = [0 for x in range(128)]

    # find frequency of words by putting them into a dictionary
    data_dict = {}
    for word in clean_data:
        data_dict.setdefault(word, 0)
        data_dict[word] += 1

    # hash words using python text hash function
    for word in data_dict.keys():
        hash_data = hashlib.md5("{0}".format(word).encode()).hexdigest()
        binary = bin(int(hash_data, 16))[2:]
        while len(binary) < 128:
            binary = '0' + binary
        k = 0
        for x in binary:
            if int(x):
                weight_vector[k] += data_dict[word]
            else:
                weight_vector[k] -= data_dict[word]
            k += 1

    binary = ''
    for v in weight_vector:
        if v < 1:
            binary += '0'
        else:
            binary += '1'
    print(binary)
    return
```

5.2 Solution

Below, it is the result of running *simhash*

Starting Time: Wed, Sep 21, 2016 at 22:11:53

10010101011111101010010111110000101101101011010001001110110000111000011110010000110110100101101101010010101110011111010101111101011001010101101101010010111110000101101101011010001001110110000111000011110010000110110100101101101010111101110111011101011111010110000100111001110111111110101000111011101001111001010100000001010010001000110010010101110101111111110001010011011011000101101000010011100111011111111010100011101110100111100101010000000101001000100011001001010111010111111111000101001101101100010110100000

End Time: Wed, Sep 21, 2016 at 22:11:53
Execution Time: 0.06 seconds

We can notice that the first two hashes are very similar, almost identical. While the third hash is very distinct from the rest.