CS-734/834 Introduction to Information Retrieval: Assignment #2: Ex 4.1, 4.2, 4.3, 4.8 & 5.10

Due on Thursday, October 12, 2016

Dr. Michael L. Nelson

Plinio Vargas
pvargas@cs.odu.edu

Contents

List of Figures

Listings

List of Tables

1 Exercise 4.1

Plot rank-frequency curves (using a log-log graph) for words and bigrams in the Wikipedia collection available through the book website (http://www.searchengines- book.com). Plot a curve for the combination of the two. What are the best values for the parameter c for each curve?

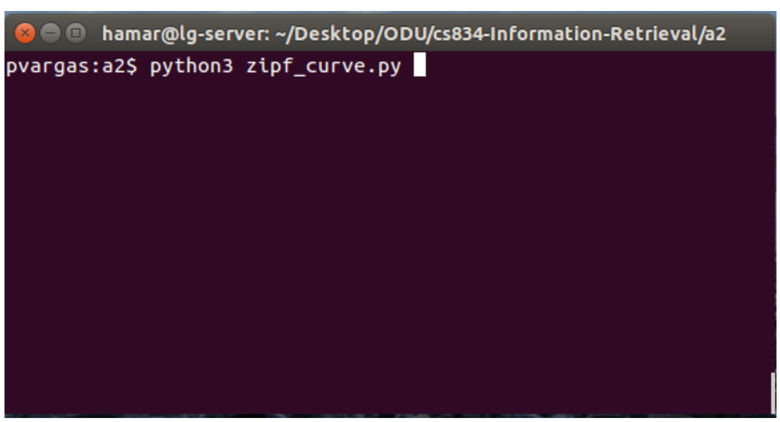
1.1 Approach

The collection available in the book's website was downloaded from `http://www.search-engines-book.com/collections/`. There are various collections within the site. Experiments were performed initially with the small collection (**Wiki small**) which contains 6,043 documents. Although plotting a **Zipf curve** will work better in large collections, the small collection helped in the testing and the debugging of our script.

The script **zipf_curve.py** was developed to tokenize the collection. It also produced the data (*zipf_law*.txt*) required to plot the curves. The "R" script **zipf_law.R** was developed to plot and find the best c for each curve.

1.1.1 Running Zipf_curve.py

Figure 1: Running zipt_curve.py from Command Line



1.1.2 Collection Tokenization

The collection was installed in folder **./en** (line 27 Listing 1). **zipf_curve.py** recursively read all documents until reaching the bottom of the tree. Since the documents are in a **HTML** format, the library **BeautifuSoup** was utilized to extract the words from the documents. Then, it followed a similar approach proposed in the textbook [1] by:

1. Transforming all words to **lowercase**. Line 74.

Exercise 4.1 continued on next page...

2. Replacing with blanks some special characters that were not bringing additional meaning to a word, such as ?, &, *, etc. Line 67. This step required running the program various times and checking the index to see which unigrams were created that did not requir to be tokenized.

Listing 1: zipf_curve.py

```
16 data_dict = {}
17 data_bigram = {}
18 total_words = 0
19 total_bigrams = 0
20 passes = 0
21
22 def main():
23     # record running time
24     start = time()
25     print('Starting Time: %s\n' % strftime("%a,  %b %d, %Y at %H:%M:%S", localtime()))
26
27     path = './en'
28
29     for filename in os.listdir(path):
30         tokenize(os.path.join(path, filename))
31
32     print(data_dict)
33     data = sorted(data_dict.items(), key=lambda x:x[1], reverse=True)
34     print(data[:20])
35
36     with open('zipf_law.txt', 'w') as f:
37         f.write('Word\t\value\n')
38         for value, key in data:
39             f.write('%s\t%s\n' % (value, key))
40
41     print(len(data_dict), passes, total_words)
42     print(len(data_bigram))
43     data = sorted(data_bigram.items(), key=lambda x:x[1], reverse=True)
44     print(data[:20])
45
46     with open('zipf_law_bigram.txt', 'w') as f:
47         f.write('Word\t\value\n')
48         for value, key in data:
49             f.write('%s\t%s\n' % (value, key))
50
51     print('\nEnd Time:  %s' % strftime("%a,  %b %d, %Y at %H:%M:%S", localtime()))
52     print('Execution Time: %.2f seconds' % (time()-start))
53     return
54
55
56 def tokenize(url):
57     global passes, total_words, data_bigram, total_bigrams
58
59     if os.path.isfile(url):
60         passes += 1
61         f = open(url, 'r')
62         page = f.read()
```

Exercise 4.1 continued on next page. . .

```
63         f.close()
64 
65         soup = BeautifulSoup(page, 'html.parser')
66         data = soup.body.get_text()
67         data = re.sub('[*#/=?&>}{!<)(;,|\"\.\[\]]', ' ', data)
68 
69         corpus = []
70 
71 
72         # include unigram
73         for unigram in data.split():
74             unigram = unigram.lower()
75 
76             # remove empty string
77             if len(unigram) > 0 and unigram != 's' and unigram != '-' and \
78                     unigram != '' and unigram != '' and unigram != '--' and \
79                     unigram != ' ':
80                 data_dict.setdefault(unigram, 0)
81                 data_dict[unigram] += 1
82                 corpus.append(unigram)
83                 total_words += 1
84                 print(unigram, end=', ')
85 
86         del data, page, soup
87 
88         # include bigram
89         n = 1
90         for words in corpus[:-1]:
91             bigram = words + ' ' + corpus[n]
92             n += 1
93             data_bigram.setdefault(bigram, 0)
94             data_bigram[bigram] += 1
95             print(bigram, end=', ')
96 
97         print()
98 
99     if not os.path.isdir(url):
100         return
101 
102     for filename in os.listdir(url):
103         tokenize(os.path.join(url, filename))
104 
105     return
```

1.1.3 Data Collection

Per requirement, the collection MUST be tokenized using *unigram* and *bigram* tokens. **zipf_curve.py** made two iterations:

1. Find the *unigrams* and their frequency within a document. Lines 72-84.

2. Once the token discrimination had been performed in the previous step, the side by side word pairing was performed in memory to obtain the *bigrams*. Lines 90-95.

The word frequency is added to its collection every time a word is extracted. The word frequency is added to a dictionary structure that contains all the words and their frequency in the collection. Lines 80 and 94.

1.2 Solution

Figure 2: Result from **zipt_curve.py** Small Collection

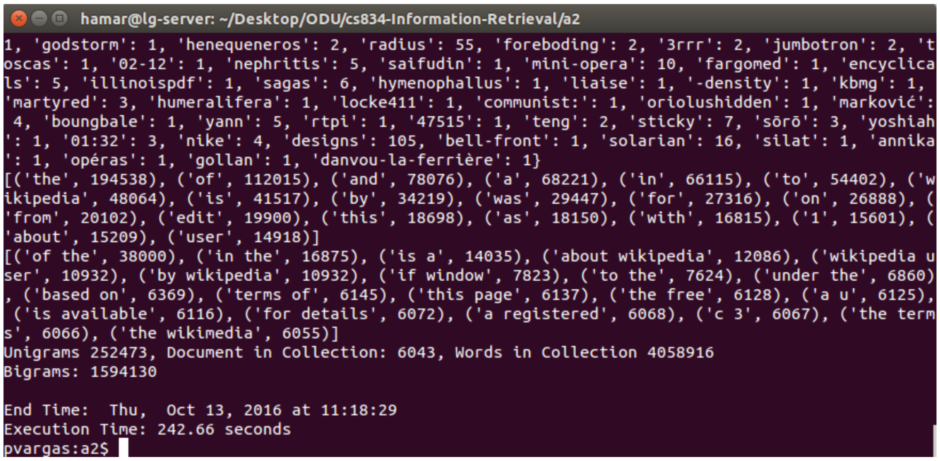


Figure 3: Result from **zipt_curve.py** Large Collection

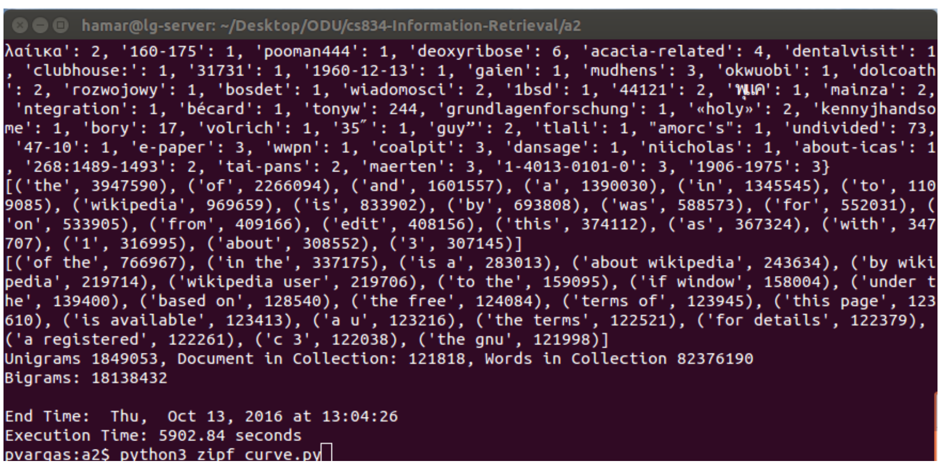
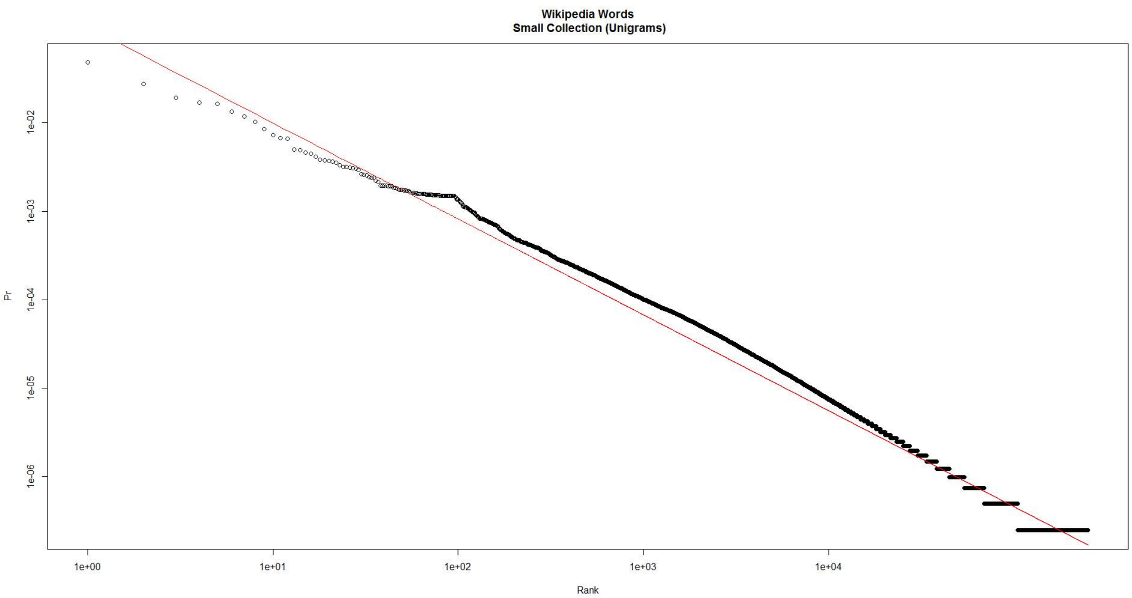


Table 1: Wikipedia Collection

	Small	Large
Total Documents	6,043	121,818
Total Word Occurrences	4,058,916	82,376,190
Number of Unigrams	252,473	1,849,053
Number of Bigrams	1,594,130	18,138,432

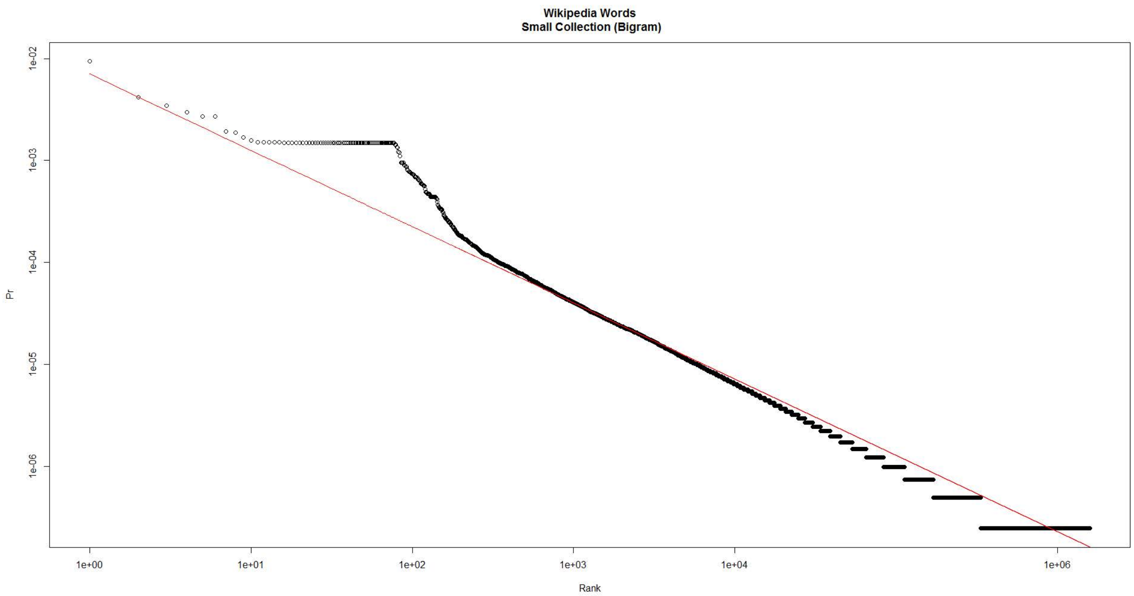
Figure 4: Unigram Zipf-Curve




```
Maximum likelihood estimation

Call:
mle(minuslogl = ll, start = list(s = 1.08007))

Coefficients:
  Estimate   Std. Error
s 1.010182 0.0001326132

-2 log L: 67871958
> exp(lzipf(s.sq,max_x))[1]
[1] 0.1204752
```

Estimation for C in Wikipedia unigram small collection is **0.1204752**

Figure 5: Bigram Zipf-Curve



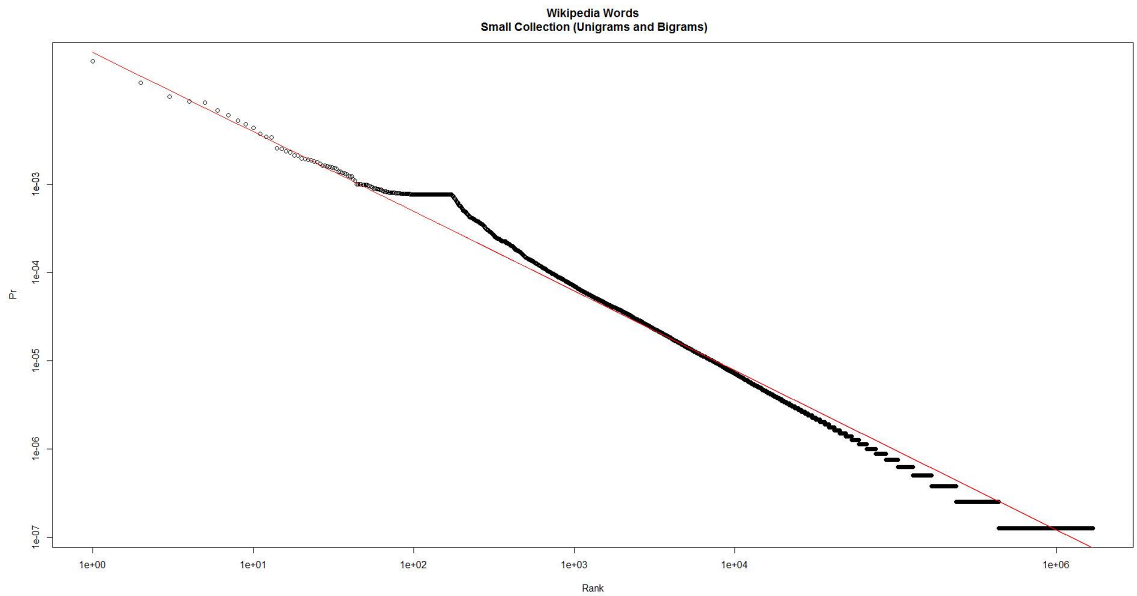

```
Maximum likelihood estimation

Call:
mle(minuslogl = ll, start = list(s = 1))

Coefficients:
   Estimate  Std. Error
s 0.8194184 0.000136322

-2 log L: 99517651
> exp(lzipf(s.sq,max_x))[1]
[1] 0.1123482
```

Estimation for C in Wikipedia bigram small collection is **0.1123482**

1.2.1 Combined Curve

We can find the value of C for the combined curves in Figure 6 by getting the coefficient for the best fitted line in the log graph. Calculation in **R** provides:

Figure 6: Combine Zipf-Curve




```
Maximum likelihood estimation

Call:
mle(minuslogl = ll, start = list(s = 1))

Coefficients:
   Estimate   Std. Error
s 0.9232369 8.383762e-05

-2 log L: 178325957
> exp(lzipf(s.sq,max_x))[1]
[1] 0.1117953
```

Making C for the combine curve = **0.1117953**

2 Exercise 4.2

Plot vocabulary growth for the Wikipedia collection and estimate the parameters for Heaps law. Should the order in which the documents are processed make any difference?

2.1 Approach

Python script *heaps_curve.py*, shown on Listing 2, was developed to complete this exercise. The previous script **zipf_curve.py** is very similar to *heaps_curve.py*. A modification was made to the previous solution to solve problem 2.

The main difference in this approach is that we tally the sum of all words in the dictionary as each document was processed, in line 71, ONLY *unigram* tokens are used.

Listing 2: heaps_curve.py

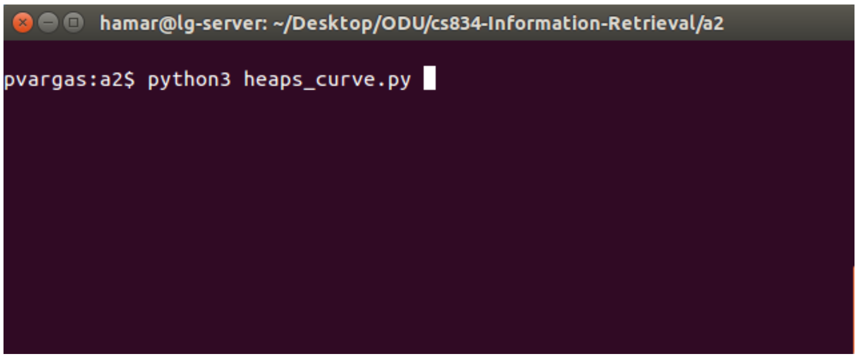
```
44 def tokenize(url):
45     global passes, total_words, data_bigram
46 
47     if os.path.isfile(url):
48         passes += 1
49         f = open(url, 'r')
50         page = f.read()
51         f.close()
52 
53         soup = BeautifulSoup(page, 'html.parser')
54         data = soup.body.get_text()
55         data = re.sub('[*#/=?&>}{!<)(;,|\"\.\[\]]', ' ', data)
56 
57         # include unigram
58         for unigram in data.split():
59             unigram = unigram.lower()
60 
61             # remove empty string
62             if len(unigram) > 0 and unigram != 's' and unigram != '-' and \
63                     unigram != '' and unigram != '' and unigram != '--' and \
64                     unigram != ' ':
65                 data_dict.setdefault(unigram, 0)
66                 data_dict[unigram] += 1
67                 total_words += 1
68 
69                 print(unigram, end=', ')
70 
71         heaps_data.append(len(data_dict))
72         del data, page, soup
73 
74     if not os.path.isdir(url):
75         return
76 
```

Exercise 4.2 continued on next page...

```
    for filename in os.listdir(url):
        tokenize(os.path.join(url, filename))

    return
```

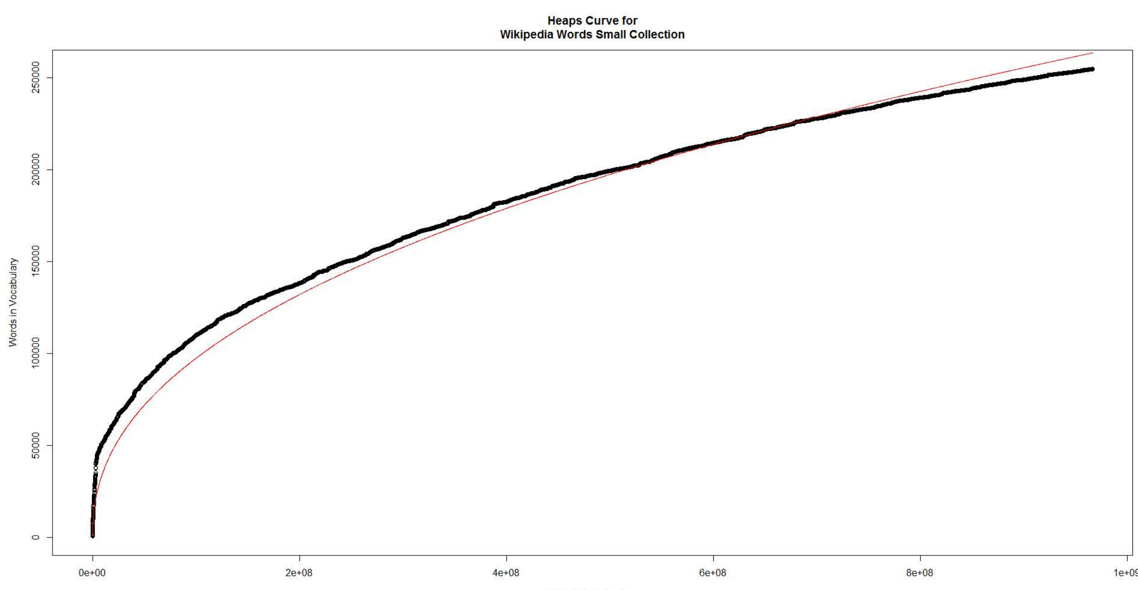
2.1.1 Running Heaps_curve.py

Figure 7: Running Heaps_curve.py from Command Line



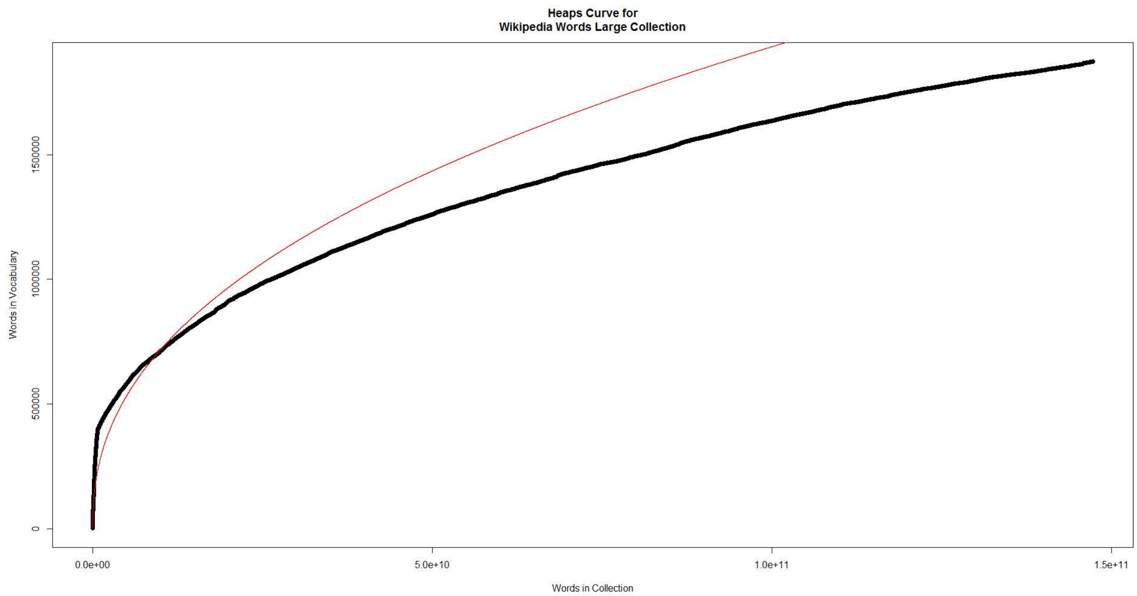
2.2 Solution

Figure 8: Heaps-Curve for Wikepedia Small Collection



The red curve represents the prediction values for Wikipedia small collection using function $v = Kn^{\beta}$, where $K = 36$ and $\beta = 0.43$. The data for this curve is on the file *heaps_curve.txt*. It was plotted using the **R** script *heaps_curve.R*

Figure 9: Heaps-Curve for Wikepedia Large Collection



The red curve represents the prediction values for Wikipedia small collection for $v = Kn^{\beta}$. The black curve was plotted using Wikipedia large collection values. Its data is contained in file *heaps_curve-l.txt*.

3 Exercise 4.3

Try to estimate the number of web pages indexed by two different search engines using the technique described in this chapter. Compare the size estimates from a range of queries and discuss the consistency (or lack of it) of these estimates.

3.1 Approach

According to the textbook [1] with the assumption that all terms occur independently:

$$f_{ab} = N \cdot f_a/N \cdot f_b/N = (f_a \cdot f_b)/N \tag{1}$$

Where:

N is the number of documents in the collection.
f_i is the number of document that term i occurs.
f_{ab} is the combined set result.

Then, we can estimate N by:

$$N = \frac{(f_a \cdot f_b)}{f_{ab}} \tag{2}$$

3.1.1 Search Engines

The two search engines used for this exercise were: **Google** and **Bing**

3.1.2 Query Terms

The query term used for this experiment was: *solar avocado*

Exercise 4.3 continued on next page...

3.2 Solution

3.2.1 Google Estimation

Looking at Figure 10:

$$f_{solar} = 623,000,000$$

Looking at Figure 11:

$$f_{avocado} = 70,700,000$$

Looking at Figure 12:

$$f_{solar\ avocado} = 562,000$$

Figure 10: Google Solar Term

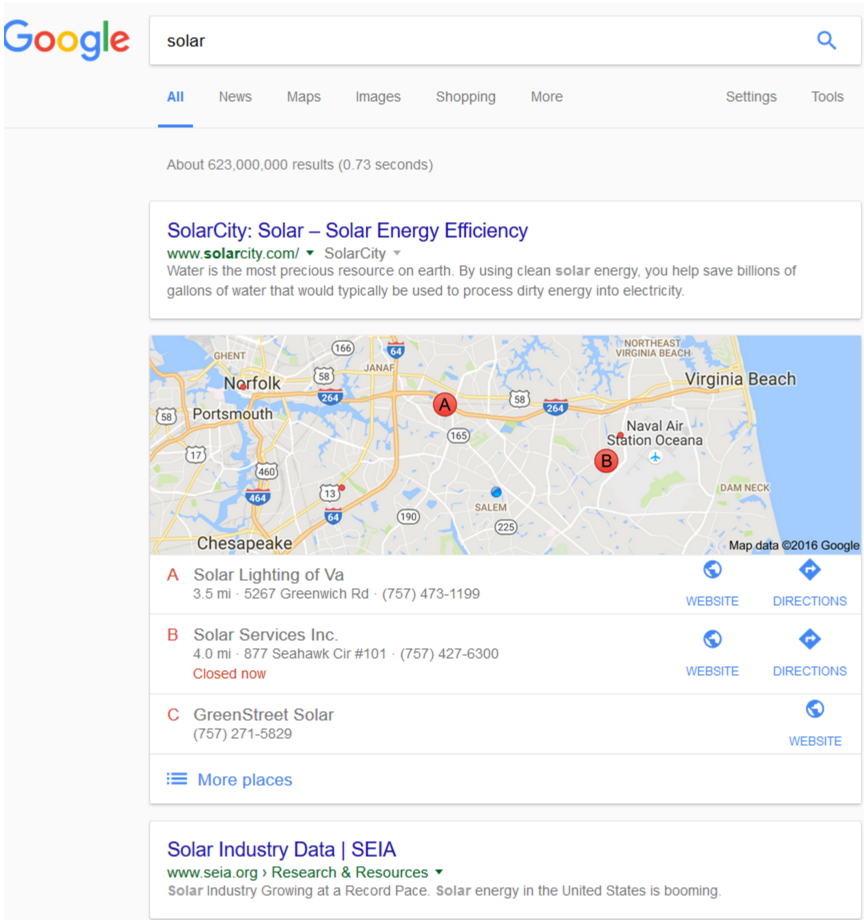


Figure 11: Google Avocado Term

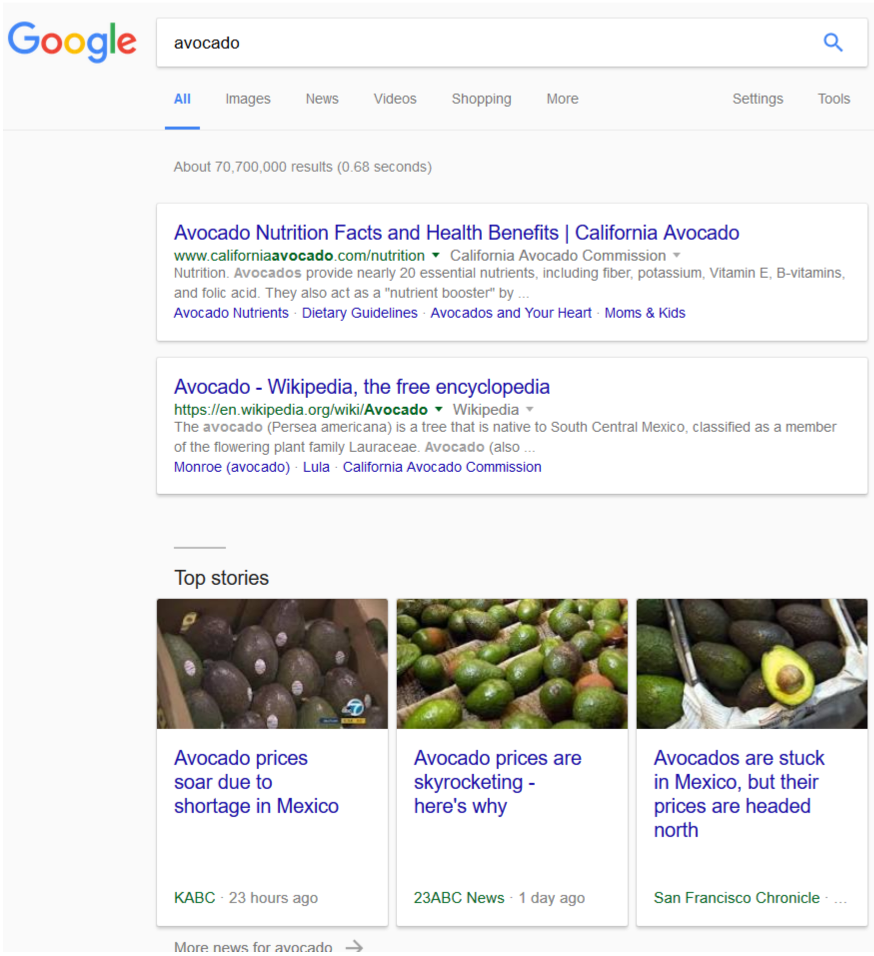
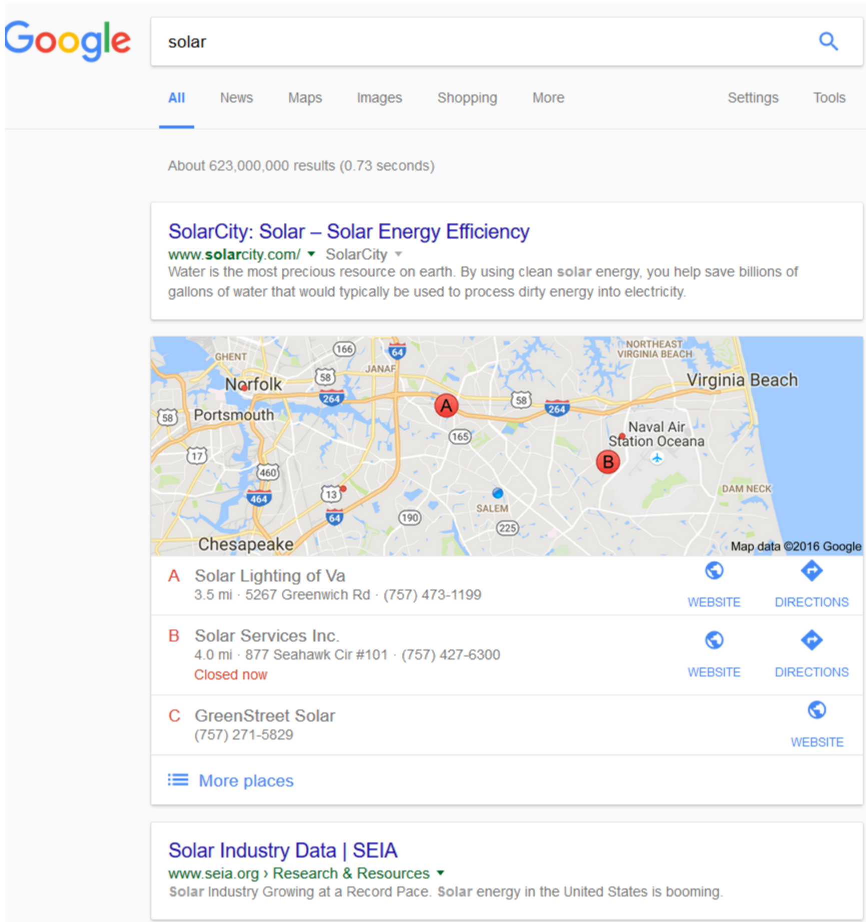


Figure 12: Google Solar Avocado Term



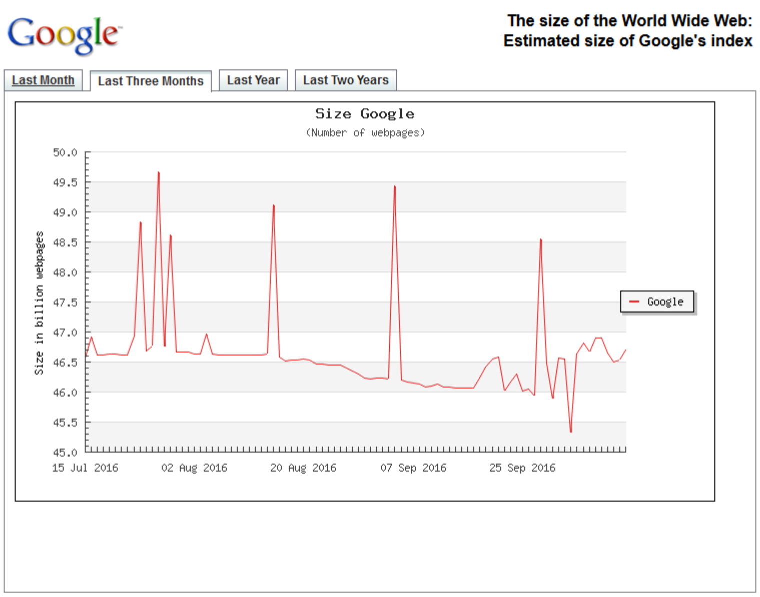
3.2.1.1 Google Estimation of N From equation (1) we can estimate the value of N by:

$$N = \frac{(623,000,000 \cdot 70,700,000)}{562,000}$$

$$N = 78.4 \ billion$$

Making a comparison with Figure 13, our estimation is 78.4/45.5 or approximately three quarters larger of the size estimated in `http://www.worldwidewebsize.com/`.

Figure 13: Google Collection Size



3.2.2 Bing Estimation

Looking at Figure 14:

$$f_{solar} = 44,900,000$$

Looking at Figure 15:

$$f_{avocado} = 13,900,000$$

Looking at Figure 16:

$$f_{solar\ avocado} = 811,000$$

Figure 14: Bing Solar Term

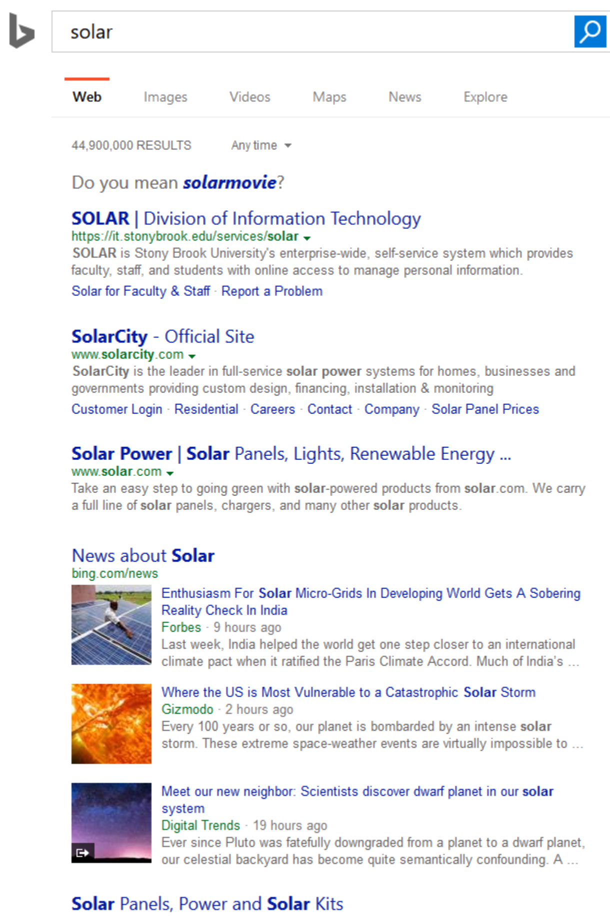
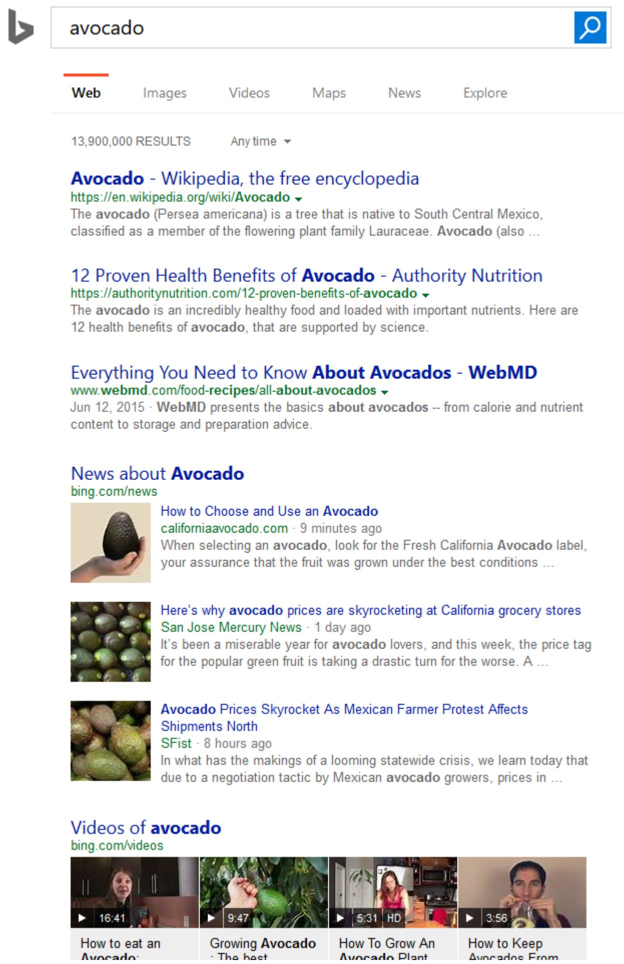
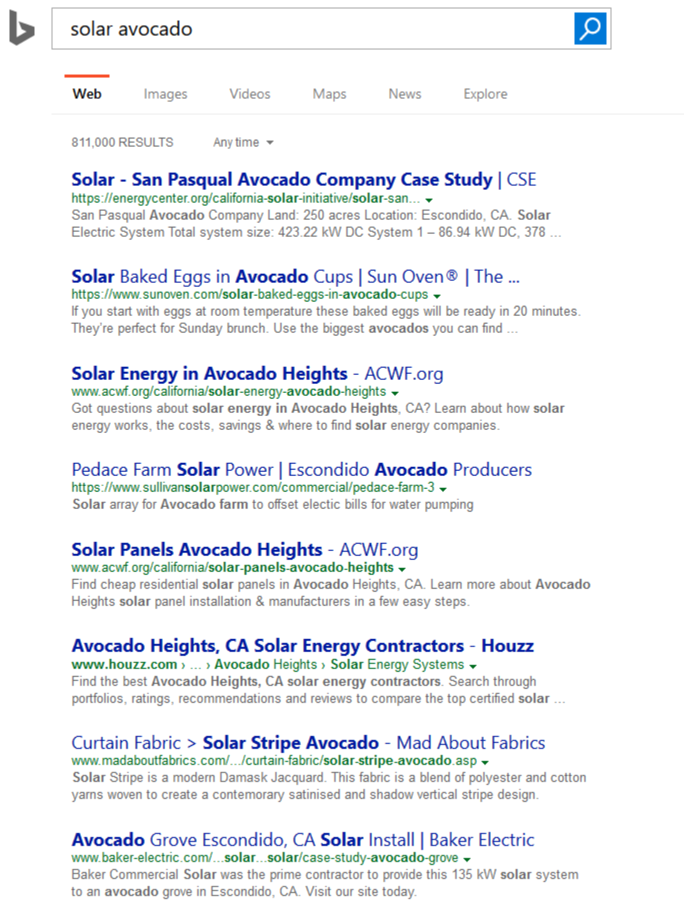


Figure 15: Bing Avocado Term



Exercise 4.3 continued on next page...

Figure 16: Bing Solar Avocado Term



Exercise 4.3 continued on next page...

3.2.2.1 Bing Estimation of N

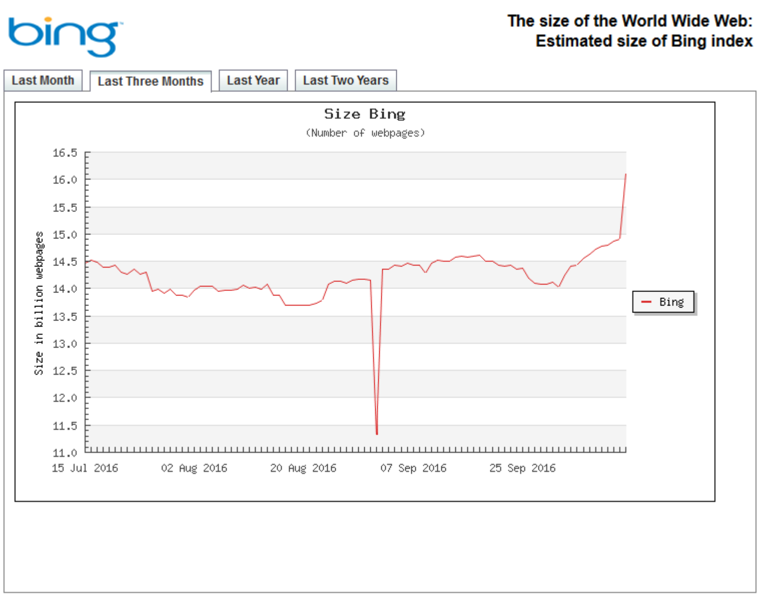
From equation (1) we can estimate the value of N by:

$$N = \frac{(44,900,000 \cdot 13,900,000)}{811,000}$$

$$N = 77\ million$$

Making a comparison with Figure 17, our estimation is 77/1600 or approximately 5 percent of the size estimated in `http://www.worldwidewebsize.com/`.

Figure 17: Bing Collection Size



4 Exercise 4.8

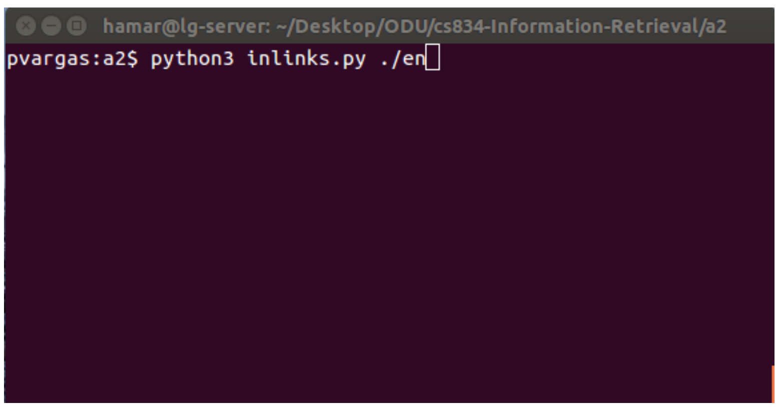
Find the 10 Wikipedia documents with the most inlinks. Show the collection of anchor text for those pages

4.1 Approach

The task was divided into two sub-problems: **counting links** and **pairing links** with actual resources. The python script *inklinks.py* (see Listing 3) was developed to solve this problem. There were many links pointing to relative non-existent links within the collection.

4.1.1 Running Inlinks.py

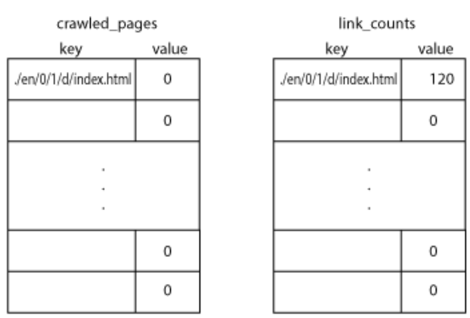
Figure 18: Running Inlinks.py



4.1.2 Counting Links

Two dictionaries are created: **crawled_pages** and **link_count**. The first keeps track of all resources retrieved from the collection. Its key is the URI of the collection. Its value is initialized to zero (line 24). All the links in the HTML page are extracted and added to second dictionary. See lines 37-45 in Listing 3.

Figure 19: Inlinks Dictionary



4.1.3 Pairing Links

Finally, the crawled pages are looked in the link counts. The value for similar keys is transferred to the **crawled_pages** dictionary (see lines 56-58 in Listing 3). This will guarantee that only valid resources will be considered for inlinks counts. The dictionary is sorted in descend order (line 61) and the top-10 inlinks are saved into file *anchor-text.txt* with their anchor texts. See lines 77-84.

Listing 3: inlinks.py

```
21 def inlink(url, crawled_pages, link_count):
22     if os.path.isfile(url):
23         # initialize link count
24         crawled_pages[url] = 0
25 
26         # read page
27         f = open(url, ’r’)
28         page = f.read()
29         f.close()
30 
31         print(’Extracting links from: %s\n’ % url)
32 
33         # create BeautifulSoup Object
34         soup = BeautifulSoup(page, ’html.parser’)
35 
36         # place source link into list
37         for link in soup.find_all(’a’):
38             uri = link.get(’href’)
39             if uri and uri[:3] == ’../’:
40                 uri = ’./en/’ + re.search(’(\.\.\/)+(.*)’, uri).group(2)
41                 anchor_text = re.search(’(.*>)(.*)(<\/a>)’, str(link))
42                 if anchor_text:
43                     print(uri, anchor_text.group(2))
44                     link_count.setdefault(uri, 0)
45                     link_count[uri] += 1
46 
47         return
48 
49     for filename in os.listdir(url):
50         inlink(os.path.join(url, filename), crawled_pages, link_count)
51 
52     return
53 
54 
55 def top_inlinks(crawled_pages, link_count, n):
56     for key in crawled_pages.keys():
57         if key in link_count:
58             crawled_pages[key] = link_count[key]
59 
60     # sort pages linked in descendant order
61     data = sorted(crawled_pages.items(), key=lambda x:x[1], reverse=True)
62 
63     with open("anchor-text.txt", ’w’) as f:
64         print(’Deleted anchort-text.txt’)
65     f.close()
66 
```

```python
    # print n anchor text
    for k in range(n):
        with open(data[k][0], 'r') as f:
            page = f.read()
        f.close()

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

        # write anchor-tags
        with open("anchor-text.txt", 'a') as f:
            f.write('%s\n' % data[k][0])
            for link in soup.find_all('a'):
                anchor_text = re.search('(.*>)(.*)(<\/a>)', str(link))
                if anchor_text and anchor_text.group(2).strip():
                    print(anchor_text.group(2))
                    f.write('%s, ' % anchor_text.group(2))
            f.write('\n\n')

        f.close()

    print(data[:10])

    return
```

4.2 Solution

The solution is the file **anchor-text.txt** uploaded under a2 folder in github.

5 Exercise 5.10

5.10. Suppose a company develops a new unambiguous lossless compression scheme for 2-bit numbers called SuperShrink. Its developers claim that it will reduce the size of any sequence of 2-bit numbers by at least 1 bit. Prove that the developers are lying. More specifically, prove that either:

- SuperShrink never uses less space than an uncompressed encoding, or
- There is an input to SuperShrink such that the compressed version is larger than the uncompressed input

You can assume that each 2-bit in put number is encoded separately.

5.1 Solution

The *SuperShrink* encryption will have to encode the following 2-bits numbers:

$$\left\{ \begin{array}{c} 00 \\ 01 \\ 10 \\ 11 \end{array} \right\}$$

The smallest sequence will have 2^4 possible scenarios:

Exercise 5.10 continued on next page. . .

00	00
00	01
00	10
00	11
01	00
01	01
01	10
01	11
10	00
10	01
10	10
10	11
11	00
11	01
11	10
11	11

We will need $Log_2(2^4)$ bits or 4 bits to account for each outcome in an unambiguous scheme. If we remove one bit to encode 4-bit sequence, what would be the decoding of sequence 111? it could be 1111 or 0111 if we truncate the head. It could be 1111 or 1110 if we truncate the tail.

References

[1] T. S. W.B. Croft, D. Metzler, *Search Engine Information Retrieval in Practice*, Pearson Education, 2015.