Universitat Pompeu Fabra Barcelona

Web Crawling and Search Engine

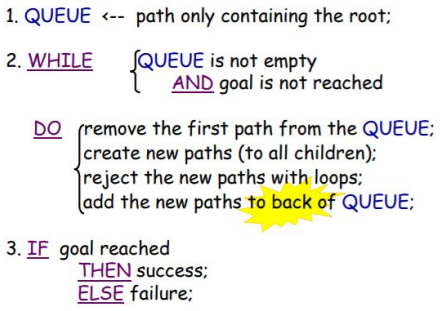
Web Intelligence

Elif Hangül
Hamit Kavas
Jonatan Koren

Web Crawling

We chose to focus on ‘Wikipedia’ as a source for our links, since it is powerful in terms of linking between pages - internal (and external links).

Breadth-First Search



First, we initialize a queue for the BFS. Moreover we define max depth and maximum of urls to crawl in order to control the search size.
The depth starts with a value of 1, we append urls to the queue and the crawler checks if it reached its limit of pages to crawl or the maximum depth search limit that we defined before, if not it will keep crawling urls.

```
# BFS: 1) remove/pop the 1st from the Queue
current_url = queue.popleft()
# Try to avoid limited server capacity ( Frequency of scraping )
time.sleep(3)
# Getting Source Code by requests package
source_code = requests.get(current_url).text
# Parse HTML by BeautifulSoup package
soup = BeautifulSoup(source_code, "html.parser")
# Get Wiki content from div element of the HTML page
content = soup.find('div', {'id': 'mw-content-text'})
```

For each url in the queue, it pop’s the first url from the queue. We derive the source code by python ‘requests’ library and parsing it’s HTML by python ‘BeautifulSoup’ library. Then, we derive the url content from the div element of the HTML page.
In order to avoid limited server capacity (Frequency of scraping) we defined sleeping time between sessions of 3 seconds. (Crawler delay), as the robots.txt does not specify explicitly how much exactly and encourages us to use -wait option.

We respect the robots.txt of the website that we are crawling so we checked for each url in the queue to see if it is allowed to be crawled or have not crawled already.

This process was done manually according to the Wikipedia robots.txt, where some urls and elements are forbidden to some 'agents' (real users or bots).

```
# Respecting robots.txt that indicates which parts of the website you should not scrap
for item in content.findAll('a', {'title': True} and {'class': False}):
    if ('#' not in item.get('href')) and item.get('href').startswith('/wiki/') \
    and not item.get('href').startswith('/wiki/Category:') \
    and not item.get('href').startswith('/wiki/File:') \
    and not item.get('href').startswith('/wiki/Template:') \
    and not item.get('href').startswith('/wiki/Book:') \
    and not item.get('href').startswith('/wiki/Portal:') \
    and not item.get('href').startswith('/wiki/Help:') \
    and not item.get('href').startswith('/wiki/Template_talk:') \
    and not item.get("href").startswith('/wiki/Special:') \
    and not item.get("href").startswith('/wiki/Wikipedia:WikiProject_Philosophy/') \
    and not item.get('href').startswith('/wiki/Talk:'):
```

In order to be more accurate and specific in the url's for the next part we defined the crawler to re-search also the related links according to keywords we gave. For example for E-bay page:

```
# Relation to specific hashtag - fill in keyword if you want or leave null
keywords = ['e-bay','ebay','products','commerce','e-commerce',
            'online shopping','big four companies','market','voucher']

for keyword in keywords:
    # BFS: 2) Create new paths to all children
    if re.search(keyword, url, re.IGNORECASE) :

            # check if there's still elements in the Queue and the URL is unique
            # ( BFS: 3) Reject the new paths with loops )
        if len(result_urls) < max_search and url not in result_urls:

            # add the URL to to results (in order to write after to a file)
            result_urls.append(url)

            # BFS: 4) add the url to the end of the Queue
            queue.append(url)
            print("For Depth ",depth,'  --->\n#' , len(result_urls) , url)

        # if goal is acheived ---> stop
        if len(result_urls) == max_search:
            return 0
```

As part of the Breadth-First Search process, the crawler checks if it hasn't reached its limits yet, and then creates 'children' to all these urls and finally adds them to the back of the queue.

Single-Page References

We found the 'References' section in wikipedia single-page to be very powerful, in terms of related urls to the same topic. Therefore, we crawled this specific section also and derived 'external links' in order to increase accuracy for the indexing part of the search engine task.

The whole process we described, is a recurring process until the crawler reaches its maximum number of pages or it's maximum depth.

Document Selection

As the next step after finding the most relevant number of links with our subject, we needed to elaborate our sources in order to limit the search smartly so that we could reach more accurate results in a shorter time. Basically, we wanted to select the top 20 relevant links per topic. To satisfy this need, we firstly scraped main Wikipedia pages for every 5 topics and obtained the most frequent words.

Recommender systems[1]

```
[('txt',
  [('recommender', 56),
   ('systems', 52),
   ('user', 28),
   ('filtering', 24),
   ('users', 24),
```

Alibaba[2]

```
[('txt',
  [('alibaba', 107),
   ('group', 35),
   ('china', 31),
   ('company', 31),
   ('alibabas', 24),
```

Amazon[3]

```
[('txt',
  [('amazon', 183),
   ('company', 42),
   ('products', 34),
   ('sales', 31),
   ('amazons', 30),
```

eBay[4]

```
{'txt': [('ebay', 139),
  ('company', 27),
  ('ebays', 25),
  ('sellers', 21),
  ('site', 18),
```

E-commerce[5]

```
[('txt',
  [('ecommerce', 74),
   ('online', 41),
   ('customers', 27),
   ('electronic', 23),
   ('products', 23),
   ('retailers', 22),
```

These words gave us an idea about selecting keywords that we will use to narrow our sources. We have widened these keywords by adding some arbitrary words that will be common for each topic, which will increase the diversity of searches. Keywords that are selected for each topic:

- *Recommender systems*

```
['electronic', 'recommendation', 'recommender', 'technology', 'service', 'platform', 'user', 'filtering', 'function', 'online',
'market', 'company', 'website', 'user', 'e-commerce', 'ecommerce', 'products', 'emtia', 'sellers', 'items']
```

- *Alibaba*

```
['alibaba', 'alibabas', 'group', 'jack', 'ma', 'china', 'service', 'platform', 'ceo', 'head', 'founder', 'online', 'market', 'c
ompany', 'website', 'sold', 'e-commerce', 'ecommerce', 'products', 'emtia', 'sellers', 'items']
```

- *Amazon*

```
['amazon', 'amazons', 'amazon web services', 'jeff', 'bezos', 'amazon kindle', 'books', 'amazon alexa', 'ceo', 'head', 'founde
r', 'online', 'market', 'company', 'website', 'sold', 'e-commerce', 'ecommerce', 'products', 'emtia', 'sellers', 'items']
```

- *eBay*

```
['ebay', 'ebay products', 'ceo', 'head', 'founder', 'online', 'market', 'company', 'website', 'sold', 'e-commerce', 'ecommerc
e', 'products', 'emtia', 'sellers', 'items']
```

- *E-commerce*

```
['electronic', 'retailers', 'information', 'technology', 'service', 'platform', 'definition', 'meaning', 'function', 'online',
'market', 'company', 'website', 'sold', 'e-commerce', 'ecommerce', 'products', 'emtia', 'sellers', 'items']
```

These are the proper keywords that are relevant for each topic so that their synonyms are also related to the subject. So that we created a list that contains synonyms for each keyword.

```
def get_synsets(keywords):
    synonyms = []
    syns = list()
    for i in keywords:
        syns.append(i)
        for syn in wordnet.synsets(str(i)):
            for lm in syn.lemmas():
                syns.append(lm.name())
    [synonyms.append(syn.replace("_", " ")) for syn in syns]
    return set(synonyms)
```

Then we opened all links with python's *selenium* package, stemmed and cleaned these pages to find the frequency of the synonyms list for each topic. The cumulative sum of the frequency of the keywords for each link is assigned as the score. Example output for *Amazon* can be seen below.

Link Id	Score	
0	754	https://en.wikipedia.org/wiki/Jeff_Bezos
1	737	https://en.wikipedia.org/wiki/Amazon_(company)
6	319	https://en.wikipedia.org/wiki/Amazon_Kindle
30	222	http://archive.wired.com/wired/archive/7.03/bezos_pr.html
24	221	https://www.wired.com/1999/03/bezos-3/
13	219	https://en.wikipedia.org/wiki/Amazon_Alexa
154	217	https://www.wired.com/wired/archive/7.03/bezos_pr.html

We get the Wikipedia pages as the most relevant as it has been expected.

Search Engine

In order to develop a search engine for HTML files (pages), we filled the necessary compartments of HTML Processor file which loads the stopwords into a hashset, separates the HTML files by title and body, does normalization, tokenizing, extracting stopwords and stemming to process the given text. After this process the index files are created for running the engine.

Tokenize, Normalize, Filtering and Stemming:

In the HtmlProcessor file, there are four main processes done to a given HTML: Normalization, tokenizing, filtering and stemming.
The class constructor is called with the path for stopwords.txt or null. Based on that, a class object is created and if the path is not null, the stopwords from the given file are read and put into a hashset. This stopwords.txt is provided for us by the instructor and is used for filtering.

In the parse method a given HTML is parsed into title and body parts using jsoup library.
In the processText method the given text goes through the four processes mentioned above.
First the text is normalized by turning into lowercase and deleting all non-letter characters. Then in the tokenizing step, the text is splitted into tokens and put into an arraylist. After that, filtering is done by checking if there are stopwords in the arraylist. This is done with the help of the hashset we created before. If there are stopwords, they are removed. Finally, with the help of the Stemmer.java file that was provided to us, we turn the words into stems for further processing.

HTML Processor vs Simple Processor:

We tested our search engine with both Simple Processor file which was given to us and Html Processor file which we filled.

Simple Processor

```
Loading index...done. Statistics:
  - Vocabulary: 80336 terms (2,46 MB).
  - Documents: 100 documents (1,44 KB).
  - Inverted: 2,87 MB.
  - Direct: 2,56 MB.
  - Cache: 5,2 MB.
```

Html Processor

```
Loading index...done. Statistics:
  - Vocabulary: 23612 terms (0,49 MB).
  - Documents: 100 documents (1,44 KB).
  - Inverted: 1,14 MB.
  - Direct: 1,05 MB.
  - Cache: 0,86 MB.
```

As it shown above, our processor has much less vocabulary to process, as it applies additional normalization (we also delete numbers), filtering and stemming then the Simple Processor which just does simple normalization and tokenizing. Thus the result is less vocabulary and cache size.

Application:

We picked E-commerce as our main topic and for 5 queries we extracted 20 pages each to try our search engine.
Our sub-topics are: Amazon, Ebay, Recommendation systems, Alibaba, Ecommerce.
Below you can find screenshots showing the indexing, engine running on interactive mode, batch mode output run file and the results obtained by the ireval.jar library.

Indexing: *java ti.SearchEngine index indexTest_hp html_files stop-words.txt*

```
done.
  Indexing file 05-7.html...Defining the Different Types of E-Commerce Businesses
done.
  Indexing file 05-8.html...Ecommerce Sales Topped $1 Trillion for First Time in 2012 - eMarketer
done.
  Indexing file 05-9.html...â€¢ Retail e-commerce sales CAGR 2023 | Statista
done.
...done:
  - Documents: 100 (26,81 MB).
  - Time: 48,19 seconds.
  - Throughput: 0,56 MB/s.
Running second pass...
  Updating term weights and direct index...done.
  Updating document norms...done.
...done
  - Time: 0,06 seconds.
Saving index...done.
Index statistics:
  - Vocabulary: 23670 terms (0,49 MB).
  - Documents: 100 documents (1,44 KB).
  - Inverted: 1,15 MB.
  - Direct: 1,06 MB.
  - Cache: 0,87 MB.
```

Engine on interactive mode: *java ti.SearchEngine interactive indexTest_hp*

```
Query (empty to exit): Who is the founder of Amazon?

1 (01-14): How Amazon founder Jeff Bezos went from the son of a teen mo...
...always been a rare combination *of* optim*is*tic, ideal*is*tic and v*is*ionary. Back in 2003, wh

2 (01-18): MacKenzie Bezos - Wikipedia
...s.com Endless.com Fire Phone Lexcycle Liquav*is*ta LivingSocial LoveFilm Mobipocket PlanetAll Re

3 (01-11): Expecting the Unexpected From Jeff Bezos - The New York Time...
...waste time on anything that *is*nâ€™t directly about *the* customer. â€œThatâ€™s where h*is* ego

4 (01-6): Amazon's Jeff Bezos: The ultimate disrupter - Fortune Manage...
...-- meaning it *is* not raining. Bezos holds court in a conference room in *the* Day One North bu
```

Batch mode: *java ti.SearchEngine batch indexTest_hp test.xml > test.run*

```
Loading index...done. Statistics:
  - Vocabulary: 23670 terms (0,49 MB).
  - Documents: 100 documents (1,44 KB).
  - Inverted: 1,15 MB.
  - Direct: 1,06 MB.
  - Cache: 0,87 MB.
001     Q0      02-17   1       0.0637184372153019      sys
001     Q0      01-20   2       0.056909170188493625    sys
001     Q0      01-13   3       0.056613548755275724    sys
001     Q0      01-14   4       0.05655967555896063     sys
001     Q0      01-6    5       0.05465151816274124     sys
001     Q0      01-17   6       0.05093209105611817     sys
001     Q0      04-13   7       0.049602449149853824    sys
001     Q0      01-16   8       0.04957591304608343     sys
001     Q0      05-5    9       0.049238468815257556    sys
001     Q0      01-12   10      0.047073999843601166    sys
001     Q0      01-8    11      0.046935758790341404    sys
001     Q0      01-11   12      0.046218646079984285    sys
001     Q0      04-5    13      0.04599356585353582     sys
001     Q0      05-2    14      0.045829410820148644    sys
```

Evaluation:

For evaluation, we created a qrel file consisting of the relevance scores of each html based on the topic and with the assistance of ireval.jar library .qrel file we created and .run file that obtained from batch mode are compared to get the precision results. For this comparison the below line is ran from the command line:
java -jar ireval.jar test.run test.qrel
Below you can see the results gathered by this process for both SimpleProcessor and HtmlProcessor.

results_simple_processor.txt

num_q	all	5
num_ret	all	279
num_rel	all	103
num_rel_ret	all	93
P@5	all	0.8400
P@10	all	0.8400
P@15	all	0.6933
P@20	all	0.6500
P@30	all	0.5133
P@50	all	0.3480
P@100	all	0.1860
P@200	all	0.0930
P@500	all	0.0372
R@5	all	0.2067
R@10	all	0.4137
R@15	all	0.5093
R@20	all	0.6355
R@30	all	0.7486
R@50	all	0.8425
R@100	all	0.9002
R@200	all	0.9002
R@500	all	0.9002
iP@0.00	all	0.9111
iP@0.10	all	0.9111
iP@0.20	all	0.9111
iP@0.30	all	0.8342
iP@0.40	all	0.8334
iP@0.50	all	0.7502
iP@0.60	all	0.7033
iP@0.70	all	0.6679
iP@0.80	all	0.5792
iP@0.90	all	0.3343
iP@1.00	all	0.1171
RP	all	0.6394
RR	all	0.8667
AP	all	0.6829
nDCG@5	all	0.7059
nDCG@10	all	0.7145
nDCG@15	all	0.6724
nDCG@20	all	0.6694
nDCG@30	all	0.7219
nDCG@50	all	0.7723
nDCG@100	all	0.7939
nDCG@200	all	0.7939
nDCG@500	all	0.7939

results_html_processor.txt

num_q	all	5
num_ret	all	450
num_rel	all	103
num_rel_ret	all	98
P@5	all	0,7200
P@10	all	0,7000
P@15	all	0,7333
P@20	all	0,6700
P@30	all	0,5200
P@50	all	0,3520
P@100	all	0,1960
P@200	all	0,0980
P@500	all	0,0392
R@5	all	0,1775
R@10	all	0,3466
R@15	all	0,5425
R@20	all	0,6551
R@30	all	0,7614
R@50	all	0,8526
R@100	all	0,9498
R@200	all	0,9498
R@500	all	0,9498
iP@0,00	all	0,8659
iP@0,10	all	0,8659
iP@0,20	all	0,8325
iP@0,30	all	0,7911
iP@0,40	all	0,7911
iP@0,50	all	0,7902
iP@0,60	all	0,7018
iP@0,70	all	0,6468
iP@0,80	all	0,5638
iP@0,90	all	0,3950
iP@1,00	all	0,1116
RP	all	0,6717
RR	all	0,7000
AP	all	0,6299
nDCG@5	all	0,4984
nDCG@10	all	0,5460
nDCG@15	all	0,6307
nDCG@20	all	0,6405
nDCG@30	all	0,6754
nDCG@50	all	0,7164
nDCG@100	all	0,7597
nDCG@200	all	0,7597
nDCG@500	all	0,7597

- Precision@k (P@k) is the proportion of recommended items in the top-k set that are relevant. The Simple Processor performs better results for the first 15 results P@15 and then the Html Processor is performing better results for higher P@k measurements. In the long term we can see Html Processor handles better.
- The R@k is very high for both processor types: the Html Processor performs almost 95% and the Simple Processor performs more than 90% of relevant pages out of the first 100 pages that we have indexed.
  For the first 30 results R@30, the Simple processor performs better while for more pages, Html Processor gets higher scores.