Search Engines

Gertjan van Noord

September 2, 2024

About the course

Information about the course is available from Brightspace

Goals of the course

- learn about search engines (duh!)
- some serious programming
  - No longer: as long as it more or less works, it is ok
    - Now: it has to work in the right way
    - by taking into account huge data sets
    - consider appropriate data-structures and algorithms
    - according to the specifications
    - simple, neat, standard, professional

Search Engines / Information Retrieval (IR)

Individuals, administrations, organizations have lots of digital information

- how to organize and store it?
- how to retrieve documents?
- how to retrieve info inside them?

An IR system is a tool to facilitate retrieval of such information

Search Engines / Information Retrieval

Here is what the book says:

Information retrieval is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers).

Search Engines / Information Retrieval

Important aspects of this definition:

- finding material
- large collections
- unstructured, usually text
- information need

Finding material

- documents
- part of documents
- tweets
- facts (e.g., what is the Dutch name for “peregrine falcon”)
- . . .

Large collections

- **Large**
- world-wide-web or parts of it
- Twitter
- specific collections (legal, medical, . . .)
- your own computer
- information within a company (manuals Boeing)
- All articles from Wikipedia

Large collection: Dutch tweets

- Collected since 2010
- One day: almost half a million tweets
- Using 'grep' to find a word in the tweets of a single day takes 20 seconds
- . . .

Unstructured

- arbitrary text versus databases
- even text is always structured somewhat
- beyond text: image, sound, video (not in this course)

Database search versus Information Retrieval

- structured versus unstructured
- search of fields versus full text search
- exact search versus inexact
- order of results alfanumerical versus order of results by "quality" (. . .)

Information Need and Evaluation

Search engines produce often a lot of results

When are you satisfied with the results?

How can we evaluate a system?

Evaluation: Precision and recall

Key statistics for evaluation with a test set (fixed questions, set of documents, evaluations of documents for the queries available).

Precision: what fraction of the system results are relevant?

Recall: what fraction of the relevant documents in the collection were returned by the system?

Information Need and Evaluation

Precision and Recall are widely used evaluation metrics.

It depends on your information need if these metrics are really appropriate.

Course objectives

- IR terminology
- IR models and IR processes
- Understanding of indexing methods, querying, retrieving, ranking, evaluation
- Practical experience by implementing basic techniques in Python
- Learn about program efficiency: what happens in case you apply particular algorithms **for very large datasets**

Chapter 1: Boolean retrieval

The first IR systems were Boolean systems.

Queries are formulated with:

- the Boolean operators AND, OR and NOT, and
- search $terms$ (for now: words)

The system should return all documents that satisfy the query.

Chapter 1: Boolean retrieval

Queries are formulated with the Boolean operators AND, OR and NOT and terms.

The system should return all documents that satisfy the query.

Examples:

- Brutus AND Caesar
- (Brutus OR Caesar) AND NOT Cleopatra
- Brutus OR (Caesar AND NOT Cleopatra)
- NOT Brutus

Information from documents

- Each document in the collection needs a unique identifier
- Each document is **tokenized**. Tokenization is the process of splitting a text into separate tokens (terms, words). This is harder than you might think.
- For a boolean system: we need to know which terms are present in which document

Term document incidence matrix

For a boolean system: we need to know which words (terms) are present in which document.

For a given query, it would be too slow to scan all documents for the given terms.

Efficient data-structure which contains the relevant information

	Doc1	Doc2	Doc3	Doc4
Antony	1	1	0	0
Brutus	1	1	1	0
Caesar	1	1	0	1
Cleopatra	1	0	0	0

What to do for a query such as: Antony AND Brutus AND NOT Cleopatra?

Term document incidence matrix

For a huge collection, this matrix becomes very big (how big?).

How big?

Example: Dutch Wikipedia.

- 256 Million words, 16 Million sentences, almost 5 Million articles.
- Number of "different" words. Types. Vocabulary. 3,5 Million (!).
- The term document incidence matrix would contain 5 Million $\times$ 3,5 Million bits, i.e., more than 2 terrabytes.

Term document incidence matrix

For a huge collection, this matrix becomes very big (how big?).

What would the row for Cleopatra look like in a real system?

More compact representation: only represent the 1's.

Alternative representation: Inverted File

- For each term (type, word), have an ordered list of document identifiers
- The ordered list of document identifiers is called *postings list*
Antony	[Doc1,Doc2]
Brutus	[Doc1,Doc2,Doc3]
Caesar	[Doc1,Doc2,Doc4]
Cleopatra	[Doc1]

Alternative representation: Inverted File

- For each term (type, word), have an ordered list of document identifiers
- The ordered list of document identifiers is called *postings list*
-

Antony	[1,2]
Brutus	[1,2,3]
Caesar	[1,2,4]
Cleopatra	[1]

How big?

Dump of the Dutch Wikipedia.

- 256 Million words, 16 Million sentences, almost 5 Million articles.
- Number of "different" words. Types. Vocabulary. 3,5 Million (!).
- The term document incidence matrix would contain 5 Million $\times$ 3,5 Million bits, is more than 2 terrabytes.
- The length of all posting lists together: 256 Million (why?).
- 3 bytes for each document id (why?). 256 Million * 3 amounts to something like 800 megabytes.

How to use posting lists

What do we do for a query such as: Brutus AND Antony.

Brutus: [1,2,3]
Antony: [1,2]

Efficient merging algorithm if posting lists are sorted

In order to find the common elements in two ordered lists: ($intersection$)

- l1 points at the next element of list1
- l2 points at the next element of list2
- if list1[l1] = list2[l2]: add the common element to the result, and move l1 and l2 one step forward
- otherwise move either l1 or l2 one step forward (whichever is the smallest).
- stop if either list is exhausted

Efficient merging algorithm if posting lists are sorted. Example.

Example: list1: [1,6,12,14,20,21]; list2: [2,10,14,16,21]. Initially, l1 points to first element of list1; l2 points to first element of list2. The result is intially empty, of course.

```
l1   list1                list2  l2            result
==>      1                    2 <==
         6                   10
        12                   14
        14                   16
        20                   21
        21
```

Efficient merging algorithm if posting lists are sorted. Example (1)

```
l1   list1                list2  l2            result
==>      1                    2 <==
         6                   10
        12                   14
        14                   16
        20                   21
        21
```

Efficient merging algorithm if posting lists are sorted. Example (2)

```
l1    list1                list2  l2            result
          1                    2 <==
==>       6                   10
         12                   14
         14                   16
         20                   21
         21
```

Efficient merging algorithm if posting lists are sorted. Example (3)

l1	list1	list2	l2	result
	1	2		
==>	6	10	<==	
	12	14		
	14	16		
	20	21		
	21			

Efficient merging algorithm if posting lists are sorted. Example (4)

```
l1    list1                list2  l2            result
          1                    2
          6                   10 <==
==>      12                   14
         14                   16
         20                   21
         21
```

Efficient merging algorithm if posting lists are sorted. Example (5)

```
l1    list1                list2  l2            result
          1                    2
          6                   10
==>      12                   14 <==
         14                   16
         20                   21
         21
```

Efficient merging algorithm if posting lists are sorted. Example (6)

```
l1    list1                 list2  l2            result
          1                     2
          6                    10
         12                    14 <==
==>      14                    16
         20                    21
         21
```

Efficient merging algorithm if posting lists are sorted. Example (7)

```
l1    list1                list2  l2              result
           1                    2                    14
           6                   10
          12                   14
          14                   16 <==
==>       20                   21
          21
```

Efficient merging algorithm if posting lists are sorted. Example (8)

```
l1    list1                list2  l2              result
           1                    2                    14
           6                   10
          12                   14
          14                   16
==>       20                   21 <==
          21
```

Efficient merging algorithm if posting lists are sorted. Example (9)

```
l1    list1                list2  l2              result
           1                    2                    14
           6                   10                    21
          12                   14
          14                   16
          20                   21 <==
==>       21
```

Efficient merging algorithm if posting lists are sorted

To find identical elements in two *ordered* lists, the number of steps depends on the size of both lists. If the length of the two lists are x and y, then the number of steps roughly is $x + y$. The number of steps is linear in the length of the input: if the input is ten times as long, then computation takes about ten times as long.

If the lists are *unordered*, the number of steps roughly is $x \times y$. The number of steps is quadratic in the length of the input. If the input is ten times as long, computation takes about hundred times as long.

This is an important difference in case x and y become large.

x	y	x + y	x × y
3	3	6	9
10	10	20	100
1.000	1.000	2.000	1.000.000
10.000	10.000	20.000	100.000.000

Intersection of two ordered lists in Python (1): the naive way

```
def intersect(l1,l2):
    return [ obj for obj in l1 if obj in l2 ]
```

or, equivalently, without the use of list-comprehension:

```
def intersect(l1,l2):
    sol = []
    for obj in l1:
        if obj in l2:
            sol.append(obj)
    return sol
```

- You have to compare each element in l1 with each element in l2
- If those lists 11 and l2 have n and m elements respectively, then you have to perform n $\times$ m comparisons
- You do not use the information that the lists are ordered

- Can we do better? Yes of course.

Intersection of two ordered lists in Python (2): merge

```
def intersect(l1,l2):
    sol = []
    try:
        it1 = iter(l1)
        it2 = iter(l2)
        n1 = next(it1)
        n2 = next(it2)
        while True:
            if n1 == n2:
                sol.append(n1)
                n1 = next(it1)
                n2 = next(it2)
            else:
                if n1 < n2:
                    n1 = next(it1)
                else:
                    n2 = next(it2)
    except StopIteration:
        return sol
```

Intersection of two ordered lists in Python (2): merge

- Go through both lists, from top to bottom, "in parallel"
- Number of steps: n+m

Intersection of two ordered lists in Python (3): sets

- Use Python's set datastructure
- Set in Python: roughly equivalent to dictionary with only keys, no values
- Intersection of two sets is built-in (and very fast)
- **This is efficient as long as you can work with sets throughout; it is not recommended to convert lists to sets for the sole purpose of computing the intersection**

```
aset = set(l1)
bset = set(l2)
intersect_set = aset & bset
```

Experiment

- **timeit** module in Python library
- l1 and l2 both have N elements, intersection is empty
- timings in second for different N:

N	1000	5000	10000	20000	40000
lists (naive)	0.107	2.633	10.522	42.017	167.685
lists (merge)	0.002	0.010	0.020	0.040	0.078
lists converted to sets	0.000	0.002	0.006	0.010	0.030
sets	0.000	0.001	0.001	0.002	0.002

if . . . in

This line was the culprit:

```
if obj in l2:
```

- This is potentially expensive if l2 is a (large) list
- If l2 is another datastructure, it may be cheap
  - Python dictionary
  - Python set
  - . . .
- Careful!

Efficient Search Engine

- Goal: answer queries over a corpus very quickly
- Means: preprocess the corpus to construct good data-structure(s) (e.g. the night before)
- Result: with the pre-built data-structure, answering can be quick
- Therefore: Put as much of the processing into preprocessing, all expensive operations should be done beforehand

Assignment for week 1

Dutch Wikipedia search engine

For this course: Dutch Wikipedia

- Dump from 2011
- almost 700 thousand articles
- *We only use the first three paragraphs of each article*
- This leaves 3 million sentences (44 million words, 252Mb)

Format

```
1       Albert Speer .
1       Berthold Konrad Hermann Albert Speer ( Mannheim , 19 maart 1905 --
1       Hij raakte , onder andere vanwege hun gemeenschappelijke belangste
1       Speer wordt wel de ’ architect van het Derde Rijk ’ genoemd en hij
1       Hij had zes kinderen , van wie er één , Albert Speer jr.
1       ( geboren in 1934 ) , na de oorlog eveneens een bekend architect i
1       Voor 1933 .
2       Andre Agassi .
2       Andre Kirk Agassi ( Las Vegas , 29 april 1970 ) is een voormalig t
2       Zijn vader is afkomstig uit Iran en van Assyrische en Armeense afk
2       In 1999 werd Agassi de vijfde speler in de geschiedenis van de spo
2       In totaal won hij acht grand slam toernooien en staat hiermee op d
```

Assignment: 5 parts

A. create_database(filename)

Write a Python function create_database(filename) which builds a database on the basis of the input given by the filename. You must make a dictionary in which the identifier of the Wikipedia article is used as the key. The value is the text of the article (i.e., the string concatenation of each of its sentences). This function should be available in the file create_database.py.

B. check_database(filename)

Now write a Python program check_database.py. This program expects a command line argument which is the filename of the Wikipedia collection that you want to use. It uses the function from part A to construct the database, and then reads lines from standard input. Each line is supposed to contain an identifier, and the program prints the corresponding article to standard output. Please make sure you know what is meant by standard input and standard output.

Put this function in the file construct_index.py.

C. contruct_index(docs)

Write a Python function construct_index(docs) which builds an "inverted file". The argument docsis the dictionary created in part A. This implies that you must build a dictionary where the key is a word, and the value is the "posting list", i.e., the Wikipedia identifiers in which that word occurs.

D. query_one_word.py

Write a Python program query_one_word.py which uses the functions from part A and C. Again, the program expects the name of the Wikipedia file as a command line argument. After building the appropriate data structures using the functions from part A and C, the program reads lines from standard input. Each line is supposed to contain a single word. The program prints all articles (not just the identifier) which contain that word. Note: if used interactively, you will type a word and then get all the hits. Then you type the next word, and get those hits, etc.

Please provide continuation message(s) to standard error, to indicate when the preprocessing step is finished and the query is being evaluated. Preprocessing can be thought of an activity that we may do off-line, and this part of the process can be a bit slower. In my model solution, preprocessing for the large input file takes 20 seconds. However, answering each query should be immediate (while debugging you use the small version of the data of course).

E. query_two_words.py

Implement a variant of part D called query_two_words.py. In this case, standard input contains two words (rather than one), and your program should print all articles in which both words occur. The format of the output should be the same as for the previous exercise.

Of course, you are expected to take into account the material of the first lecture. Please think of a good algorithm and good data-structures. Solutions which involve (a variant of) naive intersection will obtain a maximum grade of 3. All computation should be done as much as possible during the preparation phase (part A and C), so that the programs for part D and E can answer the queries very quickly, once preparation is finished.

You may also combine parts D and E in a variant which takes an arbitrary number of search terms per line, and prints the documents which contain all search terms. A correct implementation of this variant will boost your grade.