Introduction

Definition
Information Retrieval (IR) is the task of finding relevant texts within a large amount of unstructured data.

- Relevant = texts matching some specific criteria.
- Examples of IR tasks: searching for emails from a given person, searching for an event that occurred on a given date using the Internet, etc.
- Examples of IR systems: www search engines, specific search engines (laws, medical documents), etc.
- NB: DataBases Management Systems (DBMS) are different from IR systems (data stored in a DB are structured!)

Conceptual model
An IR task is a 3-step process:

1. Asking a question (how to use the language to get what we want?)
2. Building an answer from known data (how to refer to a given text?)
3. Assessing the answer (does it contain the information we were seeking?)

Remarks:
- This model can loop, the assessing phase then allows to improve the quality of the answers (more on this later).
- The user may not be able to define exactly the question (i.e. characterize his/her information needs).
More cognitive considerations
Aspects that have to be taken into account from the IR system’s point of view:
- How to allow to ask questions about information we are not aware of?
- What if the question is ill-formed (aborting, looking for document matching close questions, asking for precisions)?
- What if there is no answer (automatic modification of the question)?
- What is the best format for the answer (single answer, ordered list of answers, etc)?
- How to be sure that the answer will be understood (vocabulary used, etc)?

All these questions have to be taken into account in the realisation of a search engine.

The big picture

NB: in this lecture, we will consider that the data are already packaged, i.e. we only focus on how to retrieve relevant documents from a given set (this is called ad hoc retrieval).

Some terminology
- An IR system looks for data matching some criteria defined by the users in their queries.
- The language used to ask a question is called the query language.
- These queries use keywords (atomic items characterizing some data).
- The basic unit of data is a document (can be a file, an article, a paragraph, etc.).
- A document corresponds to free text (may be unstructured).
- All the documents are gathered into a collection (or corpus).

Example:
1 million documents, each counting about 1000 words if each word is encoded using 6 bytes:
\[ 10^6 \times 1000 \times 6/1024 \approx 6\text{GB} \]

1. Indexing documents
Indexing documents (1 / 4)

- how to relate the user’s information need with some documents’ content?
- idea: using an index to refer to documents
- Usually an index is a list of terms that appear in a document, it can be represented mathematically as:

$$index : doc \mapsto \{ \cup_{j} keyword_{j} \}$$

- Here, the kind of index we use maps keywords to the the list of documents they appear in:

$$index' : keyword_{j} \mapsto \{ \cup_{i} doc_{i} \}$$

we call this an inverted index.

Indexing documents (2 / 4)

Example of an inverted index:

```
kw_1 \mapsto doc_1 doc_2 doc_3
kw_2 \mapsto doc_7 doc_9 ...
```

- The set of keywords is usually called the dictionary (or vocabulary).
- A document identifier appearing in the list associated with a keyword is called a posting.
- The list of document identifiers associated with a given keyword is called a posting list.

Indexing documents (3 / 4)

- Arising questions: how to build an index automatically? What are the relevant keywords? (more on this later)
- Some additional desiderata:
  - fast processing of large collections of documents,
  - having flexible matching operations (robust retrieval),
  - having the possibility to rank the retrieved document in terms of relevance.
- to ensure these requirements (especially fast processing) are fulfilled, the indexes are computed in advance.

Note that the format of the index has a huge impact on the performances of the system.

Indexing documents (4 / 4)

NB: an index is built in 4 steps:

1. Gathering of the collection (each document is given a unique identifier)
2. Segmentation of each document into a list of atomic tokens (referred to as the tokenization operation).
3. Linguistic processing of the tokens in order to normalize them (e.g. lemmatizing).
4. Indexing the documents by computing the dictionary and lists of postings.
Example: the boolean model.

The boolean model - index construction (1 / 4)

→ A first model of IR technique to build an index and apply queries on this index.

Example of input collection (Shakespeare’s plays):

Doc1
I did enact Julius Caesar:
I was killed i’ the Capitol;
Brutus killed me.

Doc2
So let it be with Caesar. The
noble Brutus hath told you Caesar
was ambitious

The boolean model - index construction (2 / 4)

First we build the list of pairs (keyword, docID):

<table>
<thead>
<tr>
<th>Keyword</th>
<th>docID</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>did</td>
<td>1</td>
</tr>
<tr>
<td>enact</td>
<td>1</td>
</tr>
<tr>
<td>Julius</td>
<td>1</td>
</tr>
<tr>
<td>Caesar</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>was</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
</tr>
<tr>
<td>i’</td>
<td>1</td>
</tr>
<tr>
<td>the</td>
<td>1</td>
</tr>
<tr>
<td>Capitol</td>
<td>1</td>
</tr>
<tr>
<td>Brutus</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
</tr>
<tr>
<td>me</td>
<td>1</td>
</tr>
</tbody>
</table>
| ...     | ...

The boolean model - index construction (3 / 4)

Then the lists are sorted by keywords, frequency information is added:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>docID</th>
<th>freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>ambitious</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>be</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Brutus</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Brutus</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Capitol</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Caesar</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Caesar</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
The boolean model - index construction (4 / 4)

Multiple occurrences of keywords are then merged to create a dictionary file and a postings file:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>NumDoc</th>
<th>Totf</th>
<th>docID freq</th>
<th>docID freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>ambitious</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>be</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Brutus</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Capitol</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Caesar</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2. Processing boolean queries

- Queries using boolean connectors such as AND and OR (note that NO is more complicated).
- Consider the example contains(Brutus) AND contains(Capitol)
  1. look up for the keyword Brutus in the dictionary
     → retrieve its posting(s)
  2. look up for the keyword Capitol in the dictionary
     → retrieve its posting(s).
  3. merge the 2 posting lists according to their docID
     → the intersection are the solutions to the query.

Processing boolean queries (continued)

Merging algorithm (from [Manning et al., 07]):

```java
merge(post1, post2)
answer <- []
while (post1 != [] AND post2 != [])
    if (docID(post1) == docID(post2))
        add(answer, docID(post1))
        post1 <- next(post1)
        post2 <- next(post2)
    else
        if (docID(post1) < docID(post2))
            post1 <- next(post1)
        else post2 <- next(post2)
    endif
endwhile
return answer
```

NB: the posting lists HAVE to be sorted.
Extended boolean queries

Generalisation of the merging process:

- imagine more than 2 keywords appear in the query, example:
  - (Brutus AND Caesar) AND NOT (Capitol)
  - Brutus AND Caesar AND Capitol
  - (Brutus OR Caesar) AND (Capitol)
  - ...

- idea: consider keywords with shorter posting lists first (to reduce the number of operations).
  → use the frequency information stored in the dictionary.

Extended boolean queries (continued)

Generalised merge algorithm (from [Manning et al,07]):

```
gemMerge([t1 .. tn])
  terms <- sortByFreq([t1 .. tn])
  answer <- postings(head(terms))
  terms <- tail(terms)
  do
    list <- postings(head(terms))
    terms <- tail(terms)
    answer <- merge(answer, list)
  while (terms != []) AND (answer != [])
  return answer
```

Faster query processing (1 / 2)

- question: how to improve this list merging?
  - idea: managing skip pointers between document IDs
  - such a pointer constitutes a link to a document ID that is situated further in the posting list
  - when 2 lists are merged, these pointers make it possible to skip some sub-lists
  - Example:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>KeyID</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

Faster query processing (2 / 2)

- question: where to place skip pointers?
  - a) many skip pointers → many comparisons to perform but more likely to skip (small) sub-lists.
  - b) few skip pointers → few comparisons, but less likely to skip (large) sub-lists.
  - Possible heuristic: for a posting list of size $L$, use $\sqrt{L}$ evenly-spaced skip pointers
  - NB: with modern hardware architectures, the cost of loading large lists is bigger than the gain of merging lists by skipping some sub-lists.
Remarks about the Boolean model

- The boolean model allows to express precise queries (you know what you get, BUT you do not have flexibility → exact matches)
- Boolean queries can be processed efficiently (time complexity of the merge algorithm is linear in the sum of the length of the lists to be merged)
- Has been a reference model in IR for a long time

Assessing the retrieval

3. Assessing the retrieval

- does the system retrieve the information you were looking for?
- evaluation of the efficiency of the system
- for instance, using a reference data set, containing both a corpus and a set of queries for which we know which documents contain the expected answer
- that is, documents have been given marks (i.e. +, 0, -) with respect to some query
- in this context, a good IR system retrieve “good” documents and “good” documents only

Assessing the retrieval (continued)

- Let us consider a query \( q \). Let us call \( R_q \) the set of retrieved texts, and \( R_q^+ \), the set of relevant texts for \( q \) in the corpus.
- A first interesting measure is the proportion of relevant texts in the retrieval:
  \[
  \frac{R_q^+ \cap R_q}{R_q} \quad \text{(precision)}
  \]
- A second interesting measure is the proportion of relevant texts found with respect to all texts marked relevant for the given query:
  \[
  \frac{R_q^+ \cap R_q}{R_q^+} \quad \text{(recall)}
  \]
**Assessing the retrieval**

**Example**

Let us consider the following documents and marks for the query "Henry":

1. Henry AG
2. Thierry Henry – FC Barcelona
3. Henry Miller – writer
4. Hotel Henry
5. Henry’s goal

- System 1: retrieve documents (2) and (4), recall ? precision ?
- System 2: retrieve documents (2), (4) and (5), recall ? precision ?

**About recall and precision**

- the recall and precision measures get their meaning with respect to a given context
  - example:
    - you are a doctor searching for some illness’ description in the web, you want all relevant texts, your priority is recall
    - you are a journalist looking for a sport result in the web, you are mainly interested in having some relevant texts only, your priority is precision
- NB: you can achieve 100 % of recall by returning all documents, thus recall is not useful without precision
- NB2: another interesting measure is the f-measure:
  \[
  2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}
  \]

**To sum up**

Today, we introduced:

- the basic notions of IR and their terminology,
- the conceptual model of an IR system,
- the boolean model for IR

In the very next lectures, we will focus on:

1. index building from raw documents
2. index compressing
3. robust query processing
4. Vector Space model

We will also have practical sessions to implement IR techniques in Java.
Assessing the retrieval

About this course

- Web page: http://www.sfs.uni-tuebingen.de/~parmenti/
  It will contain the course materials, and will be updated within the semester.
- Evaluation:
  - midterm examination (30 %)
  - final examination (40 %)
  - Java project (by pairs, 30 %)

Exercises

Exercise 1

How would you process the following queries:

- Brutus AND NOT Caesar
- Brutus OR NOT Caesar

Answer:

- Get the posting lists associated with Brutus and Caesar.
- Go through the shorter list, for each docID, if the other list contains it do not keep it in the answer list.
- New merge algorithm ...
Exercise 2

Give the general algorithm for processing the following query:

(Brutus OR Caesar) AND (Capitol OR Julius)

Answer:

◮ get the frequency of each keyword
◮ estimate the size of each OR
◮ process in increasing order of size

References

• C. Manning, P. Raghavan and H. Schütze, Introduction to Information Retrieval (chapter 1)

• C.J. van Rijsbergen, Information Retrieval, Second Edition (introduction)
  http://www.dcs.gla.ac.uk/~iain/keith/

• Searching through Shakespeare's plays:
  http://www.rhymezone.com/shakespeare