UNIVERSITÀ
DEGLI STUDI
DI MILANO

MEC ITALY

BOOLEAN RETRIEVAL

ALFIO FERRARA, STEFANO MONTANELLI

Department of Computer Science

Via Comelico 39, 20135 Milano

{alfio.ferrara,stefano.montanelli}@unimi.it
http://islab.di.unimi.it/

Outline

- Boolean retrieval as an introduction to information retrieval
- Inverted index
- Boolean queries
- From Boolean model to ranked retrieval

INFORMATION RETRIEVAL

INFORMATION RETRIEVAL

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

- Web scale: e.g., Google, Yahoo, Bing
- Personal scale: e.g., Apple's Mac OS X Spotlight, email search and spam detection
- Enterprise/Institutional scale: e.g., EU Open Data Portal, IMDB

PROBLEMS IN INFORMATION RETRIEVAL

Information retrieval is needed for:

i) Searching; ii) Classification; iii) Data analysis.

Usually, we want:

- To process large document collections quickly;
- To allow flexible matching operations, such as find **Peace NEAR love**, where **NEAR** is something like *within 5 words*;
- To allow ranked retrieval.

INDEX AND INCIDENCE MATRIX

Instead of linearly scanning all the documents in the corpus for a query, we create an **index**.

The index is resulting in a binary *term-document* **incidence matrix** C, where rows represents terms, and columns represents documents.

An element $C[i, j]$ of the matrix is 1 if the term i is contained in the document j, 0 otherwise. We will see how, in more advanced index models we can store more information in the matrix element.

EXAMPLE (SONGS)

DOCUMENT STRUCTURE


```
{
"_id" : 599,
"album" : "Blood On The Tracks",
"raw_w_count" : 434,
"title" : "Shelter From The Storm",
"url" : "http://lyrics.wikia.com/Bob_Dylan:Shelter_From_The_Storm",
"text" : "Twas in another lifetime, one of toil and blood
When blackness was a virtue the road was full of mud
I came in from the wilderness, a creature void of form
'Come in,' she said, 'I'll give you shelter from the storm'
And if I pass this way again, you can rest assured
I'll always do my best for her, on that I give my word
[...]",
"artist" : "Bob Dylan",
"len" : 2301,
"year" : 1975
}
```

EXAMPLE (SONGS)

INCIDENCE MATRIX

	Bob Dylan Shelter From The Storm	Beatles All You Need Is Love	Bob Dylan Solid Rock	Leonard Cohen Nightingale	Johnny Cash As Long As The Grass Shall Grow	Beatles Help!	...
docID	**599**	**1538**	**619**	**2221**	**4152**	**1156**	**...**
love	1	1	0	1	0	0	...
peace	0	0	1	1	1	0	...
war	1	0	1	0	1	0	...
faith	0	0	0	0	0	0	...
rock	0	0	1	0	0	0	...
please	0	0	0	0	0	1	...
know	0	1	0	0	0	1	...
...	...	...	...	...	...	...	

EXAMPLE (QUERY)

Now, processing the query

peace AND love AND NOT war

is a matter of taking the vectors of **peace**, **love**, and **war**, complement the last, and perform a bitwise **AND**

```
001110 AND 110100 AND 010101 = 000100
```

witch means the 4th document, i.e., Leonard Cohen, Nightingale

INVERTED INDEX

The main problem in working on the term-document matrix as the one shown above is that it is extremely sparse.

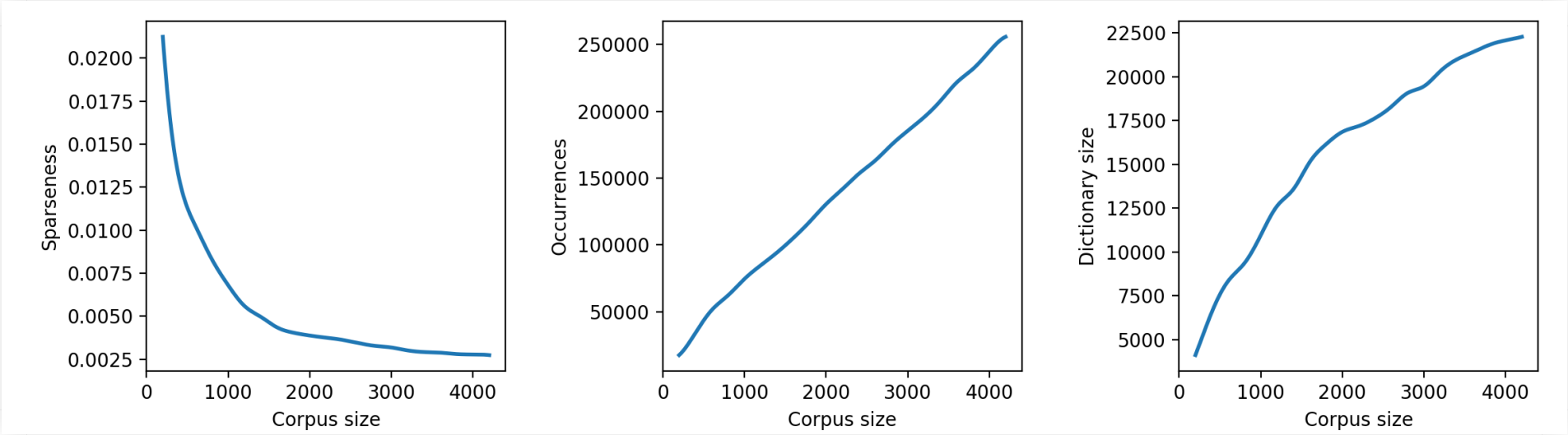
Given O, D, S as the number of token **occurrences**, the **size of dictionary** (i.e., number of unique tokens), and the **size of corpus** (i.e., number of documents), respectively, the **sparseness** $\mathcal{S}$ of a corpus can be calculated as:

$$\mathcal{S} = \frac{O}{D \cdot S}$$

Corpus	O	D	S	$\mathcal{S}$
gozzano	17,597	5,189	117	0.028985
songs	261,273	22,367	4,311	0.002710

SPARSENESS

Sparseness grows with the corpus size



The solution is to store only the positive occurrences of terms through the **inverted index**, which is one of the most important notions in information retrieval.

STRUCTURE OF THE INVERTED INDEX

Dictionary		Postings			
love	↦	599	1538	2221	...
peace	↦	619	2221	4152	...
war	↦	599	619	4152	

We note that the dictionary is sorted alphabetically, and the **posting lists** are sorted as well.

MAJOR STEPS IN BUILDING THE INVERTED INDEX

1. Tokenize text:

   `Twas in another lifetime, one of toil...` ↦ `Twas` `in` `another` `lifetime` `one` `of` `toil`

2. Linguistic preprocessing for tokens normalization:

   `Twas` `in` `another` `lifetime` `one` `of` `toil` ↦ `twa anoth lifetim one toil`

3. Give an unique ID to each document

4. Create the index and sort the posting list

(We'll see more on points 1 and 2 in the next lessons)

DB STORAGE: TOKEN-AS-TUPLE

```
{
    "token": "love",
    "postings": [599, 1538, 2221, ...]
}
```

- **Pro:**
  - Easy search: `inverted_index.find({"token": "love"})`
  - Limited cardinality of the relation/collection
- **Cons:**
  - Not available for structured models like the relational model
  - Need to keep the posting list sorted
  - Large horizontal dimension of the relation/collection

DB STORAGE: ENTRY-AS-TUPLE

```
{
    "token": "love",
    "posting": 599
}
{
    "token": "love",
    "posting": 1538
}
...
```

- **Pro:**
  - Very limited horizontal dimension of the relation/collection
  - Natural solution for relational-like DBs
  - Postings sort at query time (and/or through a DB index): `inverted_index.find({"token": "love"}).sort({"posting": 1})` or `SELECT posting FROM inverted_index WHERE token = 'love' ORDER BY posting`
- **Cons:**
  - Large cardinality of the relation/collection
  - Postings sorting could be inefficient (or require a large secondary index)

EXERCISE

Find a more efficient strategy for storing the inverted index on a DB

- Implement your solution
- Empirically demonstrate scalability and efficiency with respect to the token-as-tuple and entry-as-tuple solutions
- Discuss efficiency for create, update, delete and search the index

EXTENSIONS OF THE INVERTED INDEX

- It is a common practice to extend the inverted index in order to store more than just the existence of a token into a document
- Possible extensions include a frequency weight of the token into the document, the length of the document, and so on
- Example: suppose that the token "love" appears 3 times in the document 599 which is in turn composed by 232 tokens:

```
{
    "token": "love",
    "posting": 599,
    "occurrences": 3,
    "doc_len": 232
}
```

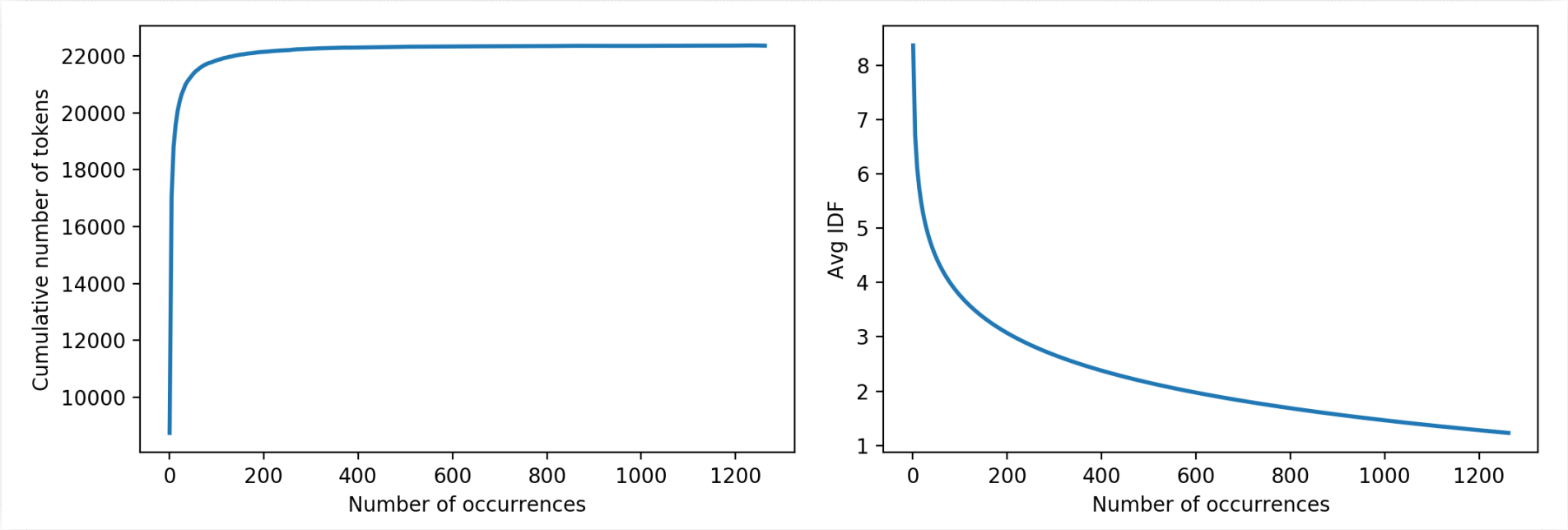
TERMS FREQUENCY

Given a token t and a document d, it is common to associate the token with the document by means of its frequency instead of just the number of occurrences.

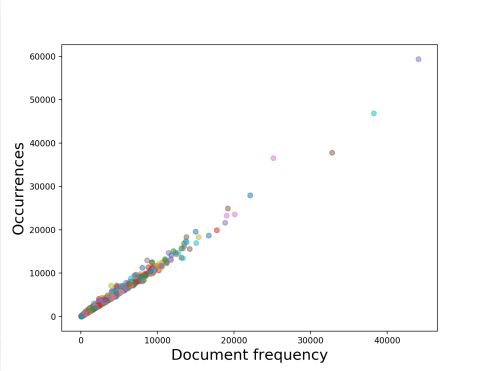
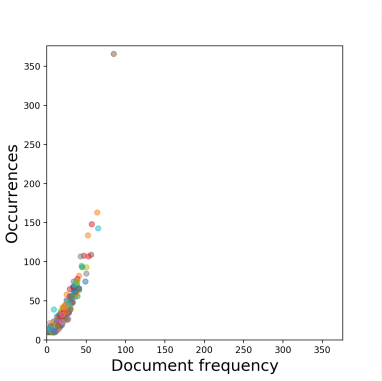
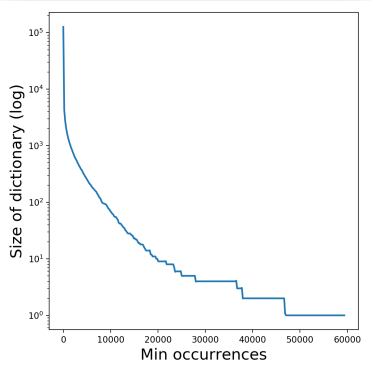
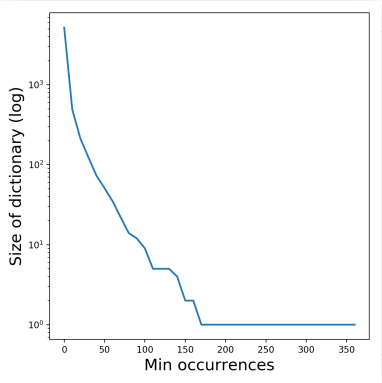
Some measures for TF (term frequency):

binary	$0, 1$
raw frequency	$o(t, d)$ of t in d
log frequency	$1 + \log o(t, d)$
normalized frequency	$\frac{o(t,d)}{\|d\|}$
log normalized frequency	$\log 1 + \frac{o(t,d)}{\|d\|}$
double normalized frequency	$K + (1 - K)\frac{o(t,d)}{\max_{\{t' \in d\}} o(t',d)}$

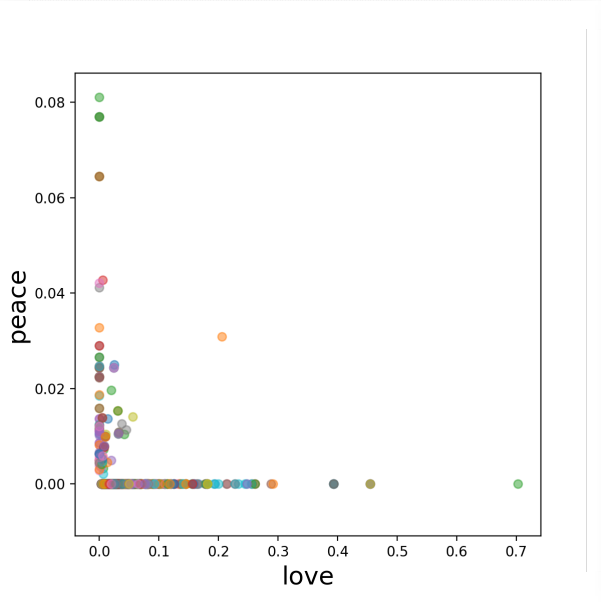
EXAMPLE (SONGS): DISTRIBUTION OF TOKENS

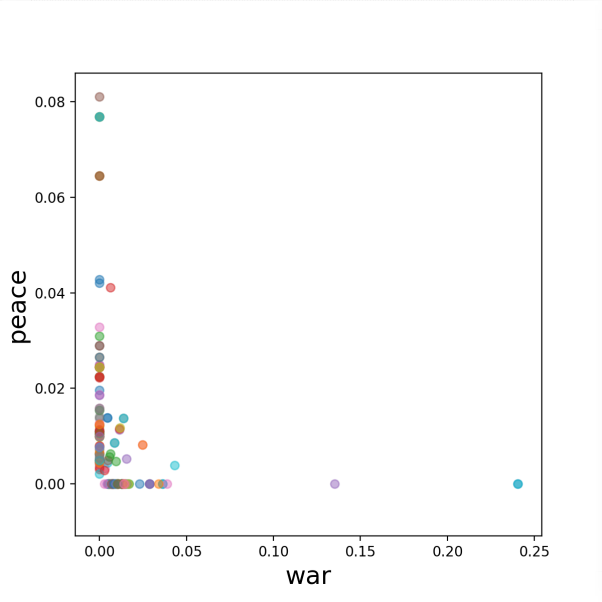


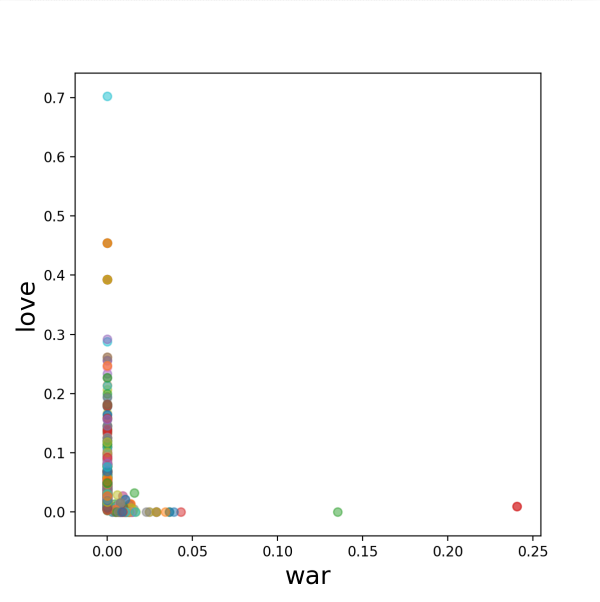
EXAMPLE (NYT / GOZZANO): DISTRIBUTION OF TOKENS

nyt	gozzano
	
	

EXAMPLE (SONGS): REPRESENTATION OF DOCUMENTS







QUERY PROCESSING

Process of the conjunctive query:

```
love AND peace
```

1. Normalize query terms with the same method used for building the inverted index
2. Locate "love" in the index and retrieve its postings
3. Locate "peace" in the index and retrieve its postings
4. Intersect the two posting lists

POSTING LISTS INTERSECTION

```
def intersection(postings_a, postings_b):
    """
    assume postings to be sorted by id
    """
    answer = []
    p1, p2 = 0, 0
    l1, l2 = len(postings_a), len(postings_b)
    while p1 < l1 and p2 < l2:
        if postings_a[p1] == postings_b[p2]:
            answer.append(postings_a[p1])
            p1 += 1
            p2 += 1
        elif p1 < p2:
            p1 += 1
        else:
            p2 += 1
    return answer
```

QUERY OPTIMIZATION

A first trivial optimization strategy for conjunctive queries is to process terms in order of increasing document frequency

Example: given the following query

```
'love' AND 'peace' AND 'war' AND 'time' AND 'captain'
```

and the frequencies: [('captain', 36), ('war', 77), ('peace', 96), ('time', 882), ('love', 1028)]

- 50 query execution with simple intersection: 0.942s
- 50 query execution with freq optimization: 0.817s

OR queries cost may be estimated by the sum of frequencies of the involved tokens

EXAMPLE WITH PYTHON

Execution of 500 random queries. Results in milliseconds.

- 0.554 intersection
- 0.524 sort intersection
- 0.752 set intersection
- 0.560 list intersection

INFORMATION NEED VS QUERY

A query is what users submit to the system to communicate their information need.

However, an information need is usually something more complex than the query and a system should try to extend and interpreting the query in order to satisfy the information need.

PRECISION AND RECALL

In order to evaluate the quality of the search results and their effectiveness, two classical measures have been defined:

PRECISION

Fraction of returned results R that are relevant to the information need, represented as the set of expected results E.

$$P = \frac{|R \cap E|}{|R|}$$

RECALL

Fraction of the set of expected results E that is retrieved by the system in the returned results R.

$$R = \frac{|R \cap E|}{|E|}$$

F-MEASURE

As a synthetic measure derived from Precision and Recall, F-measure is defined as:

$$F = 2\frac{PR}{P+R}$$

Note that the rationale of using the harmonic mean is to enforce systems to provide a good balance between Precision (all the results are effective) and Recall (all the useful documents are retrieved).

Recap

Topic	Chapters
Introduction	1
Precision and recall	1
Inverted index	1
Term frequency	1
Query processing	1