Information Retrieval

WS 2012 / 2013

Lecture 2, Wednesday October 31st, 2012

(Vector Space Model, Ranking, Precision/Recall)

Prof. Dr. Hannah Bast
Chair of Algorithms and Data Structures
Department of Computer Science
University of Freiburg

Overview of this lecture

- Organizational
  - Your experiences with Exercise Sheet 1 (inverted index)
- How to rank results
  - Vector Space Model
  - Ranking formulas
  - How to compute
  - How to evaluate
  - **Exercise Sheet 2:** compare three ranking formulas with respect to their precision and recall
- Some slides left from last lecture
  - Index construction via sorting

Experiences with ES1 (inverted index)

- Summary / excerpts   last checked October 31, 14:15
  - Setup time (SVN, Junit / Gtest, ...) for the newbies
  - Lack of programming practice for many
  - Implementation advice / header files were useful
  - In Java, need to set –Xmx=2G or more (heap size)
  - Save to file for INV would be useful ... indeed!
  - Various battles with the style checkers
  - Live programming useful / Prof. Bast coded like a beast
  - Lecture: I sat in the last row ... that sums it up for me

Ranking

- Motivation
  - **Problem:** queries often return many documents, typically more than one wants to (or even can) look at

    In web search, often millions of documents
  - **Solution:** rank the documents by relevance to the query, and show the most relevant ones first

    Plus some paging capabilities (next page of results)
  - **Problem:** how to measure relevance of a document / record to a given query?

    In a computable way, of course

Vector Space Model

all words in your collection

- Basic Idea

term-document matrix

NOTE: typically very sparse ⇒ mostly ZEROS

  - View documents (and queries) as **vectors** in a vector space
  - Each dimension corresponds to a word from the vocabulary
  - Here is an example

Document 1: uni freiburg is in freiburg
Document 2: uni karlsruhe is in karlsruhe
Document 3: freiburg is in germany

e.g. uni freiburg

	Doc 1	Doc 2	Doc 3	Query
freiburg	2	0	1	1
germany	0	0	1	0
karlsruhe	0	2	0	0
is	1	1	1	0
in	1	1	1	0
uni	1	1	0	1

Vector Space Model

- Possible entries in a document / query vector
  - **Binary** = put a 1 if the word occurs, and a 0 otherwise

    Problem: no difference between important and insigificant words

    like in the example on the previous slide
  - **Term frequency** (tf) = number of times the word occurs

    Problem: words like "the" are very frequent but carry no particular meaning

    very large for a word like „the"
  - **tf.idf** = Multiply tf with inverse document frequency

    df = number of documents containing the word

    idf = $\log_2$ (N/df), where N = total number of documents

    tf.idf = tf · idf

Vector Space Model

and always between 0 and 1 since entries are non-negative

- Similarity between two documents
  - **Cosine similarity:** $\text{sim}(d_1, d_2) = \cos \text{angle}(d_1, d_2)$

    This is 1 if vectors are the same, 0 if no word in common

    Advantage: favorable properties for mathematical analysis
  - **Dot product:** $d1 \bullet d2$ = sum of products of components

    Advantage: easy to compute efficiently ... later slide
  - From linear algebra: $d_1 \bullet d_2 = |d_1|_2 \cdot |d_2|_2 \cdot \cos \text{angle}(d_1, d_2)$
  - Therefore, if the vectors are length normalized ($|\cdot|_2 = 1$) then

    dot product = cosine similarity

$|x|_2 = \sqrt{\sum x_i^2}$

Dot product is $2 \cdot 1 + 1 \cdot 1 = 3$

$$\begin{array}{ccc} \text{Doc 1} & & \text{Query} \\ 2 & & 1 \\ 0 & & 0 \\ 0 & \bullet & 0 \\ 1 & & 0 \\ 1 & & 0 \\ 1 & & 1 \end{array}$$

Okapi BM25 1/2

number of words or number of bytes both work OK.

- BM25 = Best Match 25, Okapi = an IR system
  - This tf.idf style formula has consistently outperformed other formulas in standard benchmarks over the years

    BM25 score = $tf^* \cdot \log_2 (N / df)$, where

    $tf^* = tf \cdot (k + 1) / (k \cdot (1 - b + b \cdot DL / AVDL) + tf)$

    tf = term frequency, DL = document length, AVDL = average document length

    Often good: $k = 1.75$ and $b = 0.75$ (tuning parameters)

    Binary: $k = 0$, $b = 0$; Normal tf.idf: $k = \infty$, $b = 0$
  - There is "theory" behind this formula ... see references
  - Next slide: simple reason why the formula makes sense

Okapi BM25 2/2

- Why BM25 makes sense
  - Start with the simple formula tf · idf
  - Replace tf by tf* = tf · (k + 1) / (k + tf)
    - tf* = 0 if and only if tf = 0
    - tf* increases as tf increases ✓
    - tf* → k + 1 as tf → infinity
  - Normalize by the length of the document
    - full normalization: alpha = DL / AVDL
    - some normalization: alpha = (1 – b) + b · DL / AVDL
    - replace tf* by tf* / alpha

$$y \cdot \frac{q+1}{q+y} = \frac{q+1}{\frac{q}{y}+1}$$

$$= y \cdot \frac{1+\frac{1}{q}}{1+\frac{y}{q}} \xrightarrow{q \to \infty} y$$

if b = 0 : no normalization

BM25 is the simplest formula with these properties!

Ranking — Computation 1/4

- Conceptually:
  - For a query q and each document d, compute sim(q, d)
  - Sort the documents by sim(q, d)
  - Output the first k in the sorted sequence, for some k
  - This looks like we have to do something for each document
  - This is exactly what we wanted to avoid with an index
  - Can we also compute this based on an index?

Ranking — Computation 2/4

- Based on an index
  - In each inverted list, along with each doc id, also write the score for the corresponding word occurrence
  - Let's do this for our example, using tf.idf scores

query is always binary

	Doc1	Q
freiburg	2.0	1
germany	0	0
karlsruhe	0	0
is	...	0
in	...	0
uni	1.0	1

Document 1: uni freiburg is in freiburg
Document 2: uni karlsruhe is in karlsruhe
Document 3: freiburg is in germany
Document 4: karlsruhe is in germany

N=4

		doc id	tf.idf	doc id	tf.idf
df=2, idf=1	uni :	1,	1.0	2,	1.0
df=2, idf=1	freiburg :	1,	2.0	3,	1.0

Intersection with scores : 1, 3.0 = 1.0 + 2.0

with „union" 1, 3.0 2, 1.0 3, 1.0

Ranking — Computation 3/4

- List intersection / union with scores
  - In principle, we can use the same algorithm as before
  - But when writing a doc id to the result list, also write the **sums of the scores** from the individual lists
  - We have a choice between list "union" and "intersect"

    With intersect (so-called boolean retrieval), this computes the dot-products for docs containing all query words

    With union (so-called and-ish retrieval), this effectively computes **all** the non-zero dot-products

    so: inverted lists are an efficient way to compute matrix-vector products for sparse matrices.

Ranking — Computation 4/4

k=10

no need to recurse here

- Sorting by ranks
  - Let n be the length of the result list (# of doc ids)
  - Then a full sort would take time $\Theta(n \cdot \log n)$
  - Typically only the top-k hits need to be displayed
  - Then a **partial sort** is sufficient: get the k largest elements, for a given k with respect to similarity to the query
  - A variant of Quicksort achieves time $\Theta(n + k \cdot \log k)$
  - Running k rounds of HeapSort gives $\Theta(n + k \cdot \log n)$
  - For constant k these are both $\Theta(n)$
  - In C++ there is std::sort and std::partial_sort
  - In Java there is Collections.sort but no partial sort method

Ranking — Evaluation 1/6

- How to evaluate the quality of a ranking
  - Pick a set of queries
  - For each query, identify the **ground truth** = all relevant documents for that query

    Note: this is a very time-consuming job, especially for large document collections
  - For each query, compare the computed results list with the list of relevant documents for that query

    For the exercise sheet, just do a manual inspection of the top-10 hits

    Note that this is **not** the way to go in practice, because you have to redo that inspection after each code change

$$\frac{2}{\frac{1}{\text{prec.}} + \frac{1}{\text{recall}}}$$

Ranking — Evaluation 2/6

- Precision and Recall (ranking-unaware version)
  - Let tp = the number of relevant documents in the result list (true positives)
  - Let fp = the number of non-relevant documents in the result list (false positives)
  - Let fn = the number of relevant documents missing from the result list (false negatives)
  - Note: then tp + fp = number of documents in result list, and tp + fn = number of relevant documents
  - Then **precision** is defined as tp / (tp + fp)

    and **recall** is defined as tp / (tp + fn)
  - **F-measure** = harmonic mean of precision and recall

Ranking — Evaluation 3/6

- Precision and Recall (ranking-aware version)
  - The definitions on the previous slide are invariant under different ordering of the docs in the result list
  - Here are some ranking-aware measures

    Precision@k = the precision among the first k docs

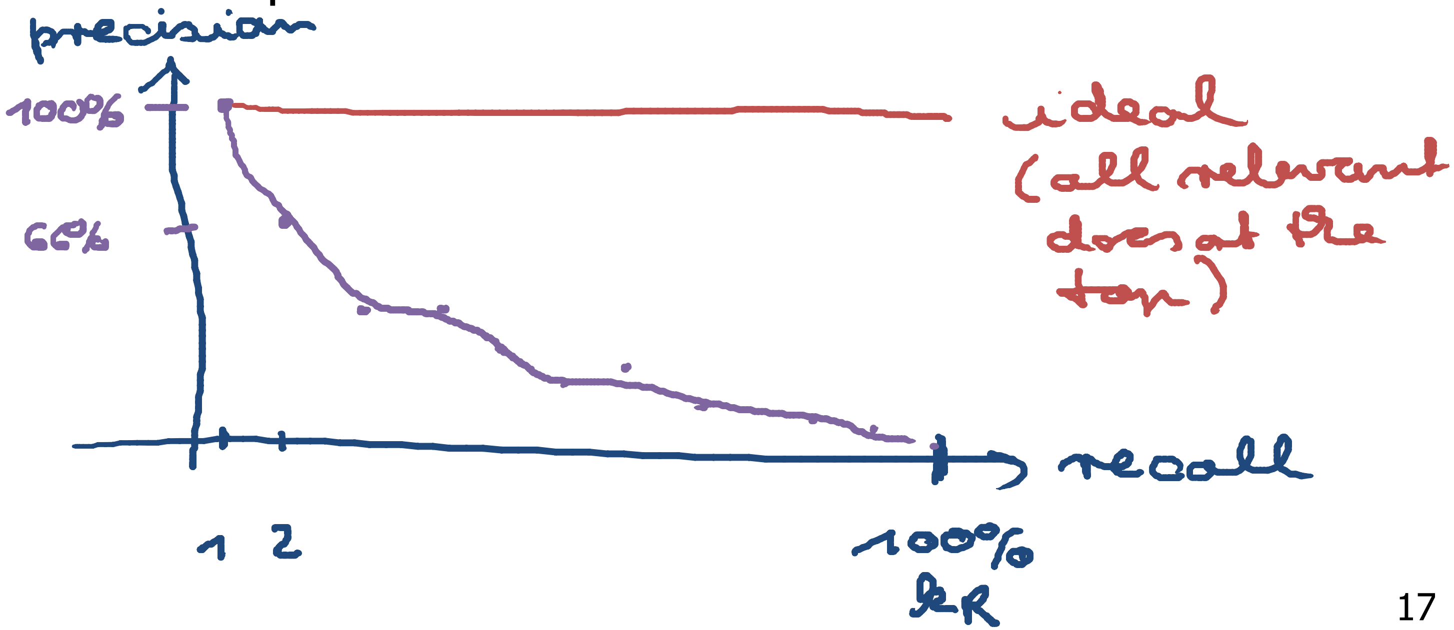
    Precision@R = the precision among the first R docs, where R is the number of relevant documents

    Let $k_1 < \ldots < k_R$ be the ranks of the relevant docs in the result list (rank missing docs randomly or worst-case)

    Average precision = average of $P@k_1, \ldots, P@k_R$
  - For a set of queries, the **MAP** = mean average precision is the average (over all queries) of the average precisions

Ranking — Evaluation 4/6

- Precision-recall curve
  - Average precision is just a single number
  - For a complete picture of the quality of the ranking, plot a precision-recall curve
  - If the x-axis is normalized, these can also be averaged over several queries



Ranking — Evaluation 5/6

- More refined measures
  - Sometimes relevance comes in more than one shade, e.g.

    0 = not relevant, 1 = somewhat rel, 2 = very relevant
  - Then a ranking that puts the very relevant docs at the top should be preferred

    relevance of the document at position i in result list

    Cumulative gain $CG@k = \Sigma_{i=1..k}\ rel_i$

    Discounted CG $DCG@k = rel_1 + \Sigma_{i=2..k}\ rel_i / \log_2 i$
  - **Problem:** CG and DCG are larger for larger result lists
  - **Solution:** normalize by maximally achievable value

    iDCG@k = value of DCG@k for **i**deal ranking

    Normalized DCG nDCG@k = DCG@k / iDCG@k

Ranking — Evaluation 6/6

- Normalized discounted cumulative gain, example

Ranking — Evaluation

- How to obtain the ground truth
  - **Method 1:** Extensive manual search

    Infeasible for very large collections
  - **Method 2:** So-called "pooling"

    Make a contest, and manually inspect only the top-k results from each participant for relevance

    Will miss relevant docs, but fair to all participants
  - **Method 3:** Crowd Sourcing

    Use services like Amazon Mechanical Turk to distribute this task over a large number of people

    Can be combined with methods 1 or 2

Implementation advice

- Index construction with tf.idf / BM25 scores
  - Elements in inverted lists now must include score

    Map<String, Array<Posting>> invertedLists;

    where Posting is a class holding a doc id and a score
  - During parse compute only basic tf: when a document contains a word multiple times, simply add 1 to the score
  - Also maintain the doc frequencies and lengths during parsing

    Map<String, int> documentFrequencies;
    Array<int> documentLengths;
  - After the parsing, go over each inverted list, and compute the final scores, e.g. BM25
  - Also see the code design suggestions on the Wiki

References

- In the Raghavan/Manning/Schütze textbook

  Section 6: Scoring, term weighting, vector space model

- Relevant Papers

  The Probabilistic Relevance Framework: BM25 and Beyond

  S. Robertson and H. Zaragoza FnTIR 2009, 333 – 389

- TREC conference (benchmarks)

  http://trec.nist.gov/tracks.html

- Relevant Wikipedia articles

  http://en.wikipedia.org/wiki/Okapi_BM25
  http://en.wikipedia.org/wiki/Precision_and_recall
  http://en.wikipedia.org/wiki/Discounted_cumulative_gain
  http://en.wikipedia.org/wiki/Partial_sorting