WARWICK

CS918: LECTURE 12

Introduction to Information Retrieval

Arkaitz Zubiaga, 12th November, 2018

LECTURE 12: CONTENTS

- What is Information Retrieval (IR)?
- Indexing Documents.
- Query Processing.
- Positional Indices.
- Other Challenges in Information Retrieval.

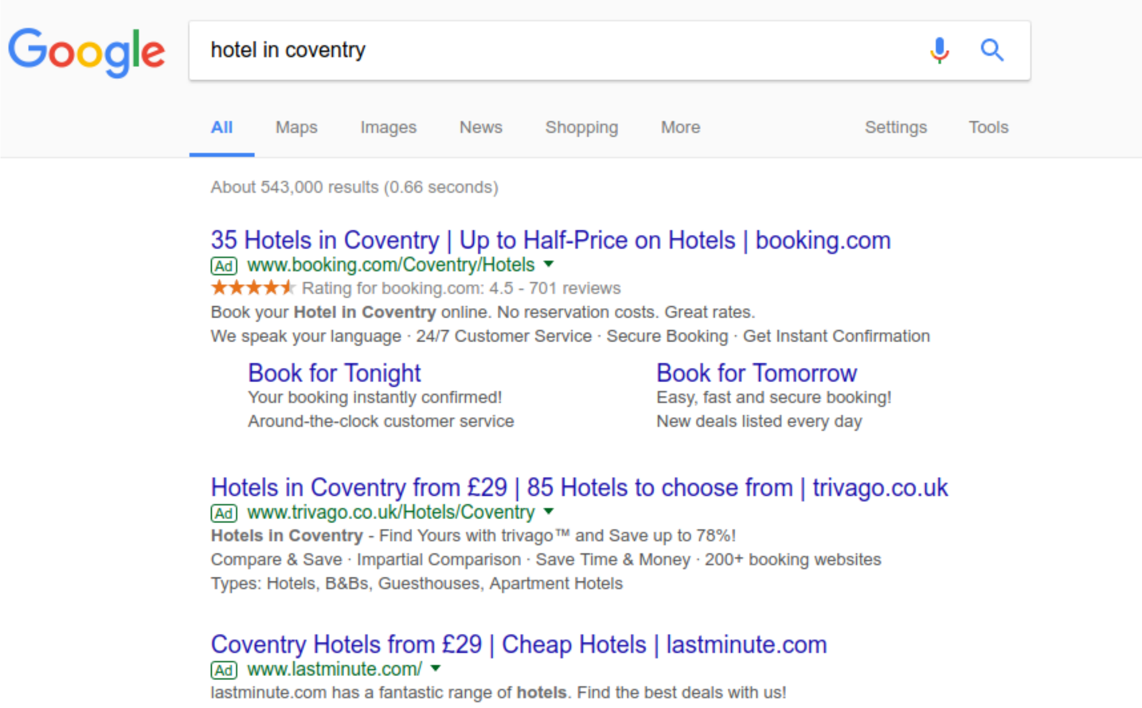
WARWICK

- **Information Retrieval (IR):** from a large collection, the task of obtaining documents that satisfy an **information need**.
  - Collection (e.g. the Web) may include images and videos.
  - In this module we'll **focus on text**.

EXAMPLES OF INFORMATION RETRIEVAL

- **Web search**, e.g. Google.
- **Vertical search**, web search on a particular topic, e.g. Yelp.
- **Email search**.
- Searching content in large **databases or hard drives**.

EXAMPLE OF AN IR SYSTEM?



INFORMATION NEED

WARWICK

- The **information need** that an IR system has to satisfy is usually expressed as a (short) text **query**, e.g. hotel coventry.
- Many **queries are vague**, i.e. the average search query has 2-4 keywords (Arantzapis & Kamps, 2008).

WARWICK

TYPES OF INFORMATION NEED

- Queries can seek **3 types of information need** (Broder, 2002):
    - **Navigational** (looking for website, e.g. Facebook).
    - **Informational** (looking for info, e.g. Thai food).
    - **Transactional** (buying/looking for product/service, e.g. iPhone 10).

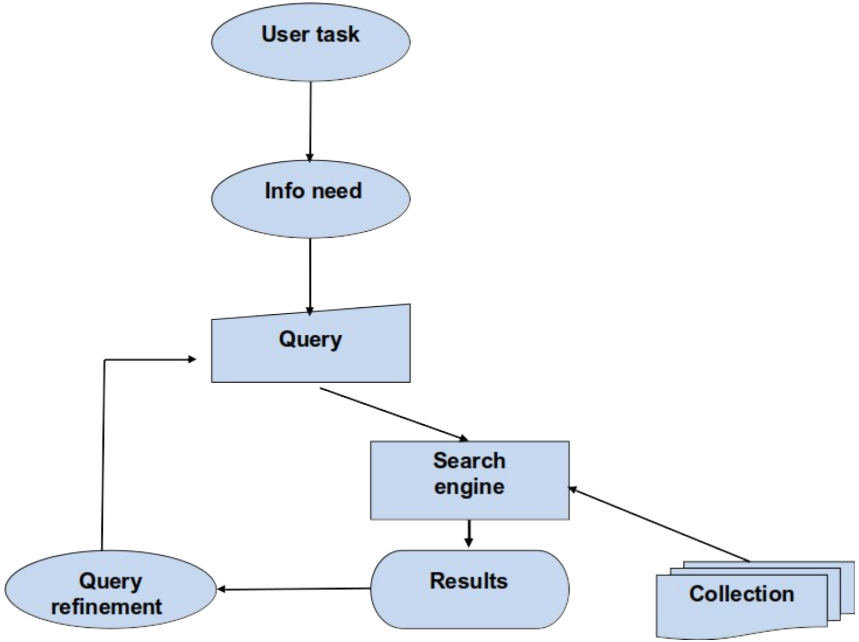
- Google redefined them as: **do, know, go**.

- **Vast majority** (up to 80%) tend to be **informational** or **know**.

THE CLASSIC SEARCH MODEL

- The **search** process **can be iterative**.
    - **Query** gives **results**.
    - If **unhappy** with results, **refine the query**.

- With a good IR system, we aim to **minimise the number of times the user refines the query**.



RELEVANCE TO A QUERY

- In IR, documents in a collection are deemed **relevant (R) or non-relevant (N) with respect to a particular query**.
  - Or sometimes levels of relevance, e.g. 1-5.

- The objective of an IR system is to present, for a given query, **as many relevant results as possible** (and as few non-relevant results as possible).
  - **58%** of users **never go to the 2nd page** of search results (Jansen et al., 1998)

SEARCHING THROUGH LARGE COLLECTIONS

- In a **small collection**, we can **process files on the fly**.
  - For a given query, go through all files and check if the query text appears in each file.
  - It would **take ages for large collections**.

- For **large collections, indexing** is the solution.
  i.e. pregenerating lists of word-document associations.

INDEXING DOCUMENTS

TERM-DOCUMENT INCIDENCE MATRICES

WARWICK

- Reduced sample of Shakespeare’s works:

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

TERM-DOCUMENT INCIDENCE MATRICES

WARWICK

- Reduced sample of Shakespeare's works:

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

- We can search for: +Brutus +Caesar -Calpurnia (2 results)

EVEN BIGGER COLLECTIONS

- Collection of **N=1M documents, each with 1K words**.
  - Say there are **M = 500K distinct terms** among these.

- 500K x 1M matrix has **half-a-trillion 0's and 1's**.
  - **Very sparse**, only one billion 1's (that's 0.2% of the values).
  - Alternative: **record only 1's** → use an **inverted index**.

INVERTED INDEX

- For each term t, **store list of documents that contain t**.
  - List needs to have **variable size**.

Brutus	1	2	4	11	31	45	173	174
Caesar	1	2	4	5	6	16	57	132
Calpurnia	2	31	54	101				

- Word ‘Brutus’ occurs in documents ID 1, ID 2, ID 4, etc.

INDEXING DOCUMENTS: EXAMPLE

WARWICK

- List (token, Document ID) pairs.

Doc 1

I did enact Julius Caesar I was killed i' the Capitol; Brutus killed me.

Doc 2

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

Term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

INDEXING DOCUMENTS: EXAMPLE

WARWICK

- Sort (alphabetically) by tokens and then doc ID.

Term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

INDEXING DOCUMENTS: EXAMPLE

WARWICK

- Merge entries + add frequency counts: **dictionary and postings**.

Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

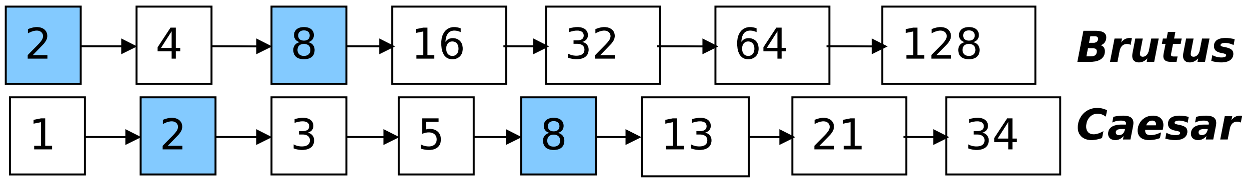
→

term	doc. freq.	→	postings lists
ambitious	1	→	2
be	1	→	2
brutus	2	→	1 → 2
capitol	1	→	1
caesar	2	→	1 → 2
did	1	→	1
enact	1	→	1
hath	1	→	2
i	1	→	1
i'	1	→	1
it	1	→	2
julius	1	→	1
killed	1	→	1
let	1	→	2
me	1	→	1
noble	1	→	2
so	1	→	2
the	2	→	1 → 2
told	1	→	2
you	1	→	2
was	2	→	1 → 2
with	1	→	2

QUERY PROCESSING

QUERY PROCESSING: AND

- Search query: **Brutus AND Caesar**
  - Retrieve postings with **Brutus**.
  - Retrieve postings with **Caesar**.
  - Get the **intersection** as the set of results (docs 2 and 8).



GETTING THE INTERSECTION

```
INTERSECT(p1, p2)
answer ← ⟨ ⟩
while p1 ≠ NIL and p2 ≠ NIL
do if docID(p1) = docID(p2)
      then ADD(answer, docID(p1))
           p1 ← next(p1)
           p2 ← next(p2)
      else if docID(p1) < docID(p2)
              then p1 ← next(p1)
              else p2 ← next(p2)
return answer
```

GETTING THE INTERSECTION

```
INTERSECT(p1, p2)
  answer ← ⟨ ⟩
  while p1 ≠ NIL and p2 ≠ NIL
  do if docID(p1) = docID(p2)
       then ADD(answer, docID(p1))
            p1 ← next(p1)
            p2 ← next(p2)
       else if docID(p1) < docID(p2)
              then p1 ← next(p1)
              else p2 ← next(p2)
  return answer
```

- Complexity of algorithm: O(m+n), as we iterate through all items – m is the length of p_1, and n is the length of p_2

SKIP POINTERS

- **Can we** improve the linear time O(m+n) and **compute it in sublinear time**?
  - We can use **skip pointers**, i.e.:
    - If p_1 has: 1, 3, 5, 15, 40
    - and p_2 has: 35, 40, 90
    - Pointers will initially be $pp_1 = 1$ and $pp_2 = 35$
    - We can skip all values smaller than 35 in p_1.

SKIP POINTERS

```
IntersectWithSkips(p1, p2)
  answer ← ⟨ ⟩
  while p1 ≠ NIL and p2 ≠ NIL
  do if docID(p1) = docID(p2)
       then Add(answer, docID(p1))
            p1 ← next(p1)
            p2 ← next(p2)
       else if docID(p1) < docID(p2)
              then if hasSkip(p1) and (docID(skip(p1)) ≤ docID(p2))
                     then while hasSkip(p1) and (docID(skip(p1)) ≤ docID(p2))
                            do p1 ← skip(p1)
                     else p1 ← next(p1)
              else if hasSkip(p2) and (docID(skip(p2)) ≤ docID(p1))
                     then while hasSkip(p2) and (docID(skip(p2)) ≤ docID(p1))
                            do p2 ← skip(p2)
                     else p2 ← next(p2)
  return answer
```

PHRASE QUERIES

- I want to search for **“University of Warwick”** – as a phrase.

- For this query, “You can live in Warwick if you are a student at the university” is **NOT a match**.

- If we want to do this search, then our **<term : docs> index is not enough**.

HOW ABOUT BIGRAM INDICES?

- **Index bigrams** instead of single words.

- For the document, “I went to the University of Warwick”, index:
  I went, went to, to the, the University, etc.

- We can **easily search for bigrams now**, but we **can’t look for “University of Warwick”** yet!
  - We can search for “University of AND of Warwick”, but there is no guarantee that they occur together in the document.

LONGER N-GRAM INDICES?

- **Longer n-gram indices are not feasible**, too many possible n-grams.
- **Bigrams** could be used when we don't need longer search queries, but it's **generally not enough**.
- Use of **bigrams to look for longer n-grams can lead to false positives** (as with the "University of AND of Warwick" example)

POSITIONAL INDICES

WARWICK

THE SOLUTION: POSITIONAL INDICES

- In the postings, **store for each term the position(s)** in which tokens of it appear:

  <term, number of docs containing term;
  doc1: position1, position2 … ;
  doc2: position1, position2 … ;
  etc.>

POSITIONAL INDEX: EXAMPLE

I have two documents:

- doc1: University of Warwick
- doc2: Warwick University

- My index:

  <University, 2; doc1: 1, doc2: 2;

  Warwick, 2; doc1: 3, doc2: 1;

  of, 1; doc1: 2>

POSITIONAL INDEX: EXAMPLE

- The search query could be: **“to be or not to be”**
  - Which documents contain it?

- For phrase queries, we **use a merge algorithm recursively at the document level**.
- But we now **need to deal with more than just equality**.

POSITIONAL INDEX: EXAMPLE

<**be**: 993427;
1: 7, 18, 33, 72, 86, 231;
2: 3, 149;
4: 17, 191, 291, 430, 434;
5: 363, 367, …>

Which of docs 1,2,4,5 could contain "***to be or not to be***"?

POSITIONAL INDEX: EXAMPLE

- Get **inverted index entries for each distinct term**: to, be, or, not.
- Merge their doc:position lists.
  - to:
    2:1,17,74,222,551; 4:8,16,190,**429,433**; 7:13,23,191; …
  - be:
    1:17,19; 4:17,191,291,**430,434**; 5:14,19,101; …
  - or…
- Look for **occurrences where the positions match the sequence** of our query: “to be or not to be”.

POSITIONAL INDEX: EXAMPLE

WARWICK

- Get **inverted index entries for each distinct term**: to, be, or, not.
- Merge their doc:position lists.
  - to:
    2:1,17,74,222,551; 4:8,16,190,**429,433**; 7:13,23,191; ...
  - be:
    1:17,19; 4:17,191,291,**430,434**; 5:14,19,101; ...
  - or...
- Look for **occurrences where the positions match the sequence** of our query: “to be or not to be”.

429: to, 430: be, 431: or, 432: not, 433: to, 434: be

BEYOND A DOCUMENT’S CONTENT

WARWICK

- What if a document doesn’t have the query keywords but it is relevant?

  e.g. if I search for “University in the West Midlands”

  warwick.ac.uk may not contain those words.

BEYOND A DOCUMENT’S CONTENT

WARWICK

- What if a document doesn’t have the query keywords but it is relevant?

Web search engines use “anchor texts” from incoming web links.

Intuition:
Warwick itself is unlikely to say “University in the West Midlands”

Another website may say “check out this university in the West Midlands”, with a link to Warwick.

BEYOND A DOCUMENT'S CONTENT

WARWICK

- OK, but what happens if an anchor text linking to Warwick says "click here".
  - Our IR system will believe Warwick is a relevant result for the "click here" search query.

- We will need to determine the **keywords** that are **meaningful**, by **weighting** them.
  - Forthcoming lectures.

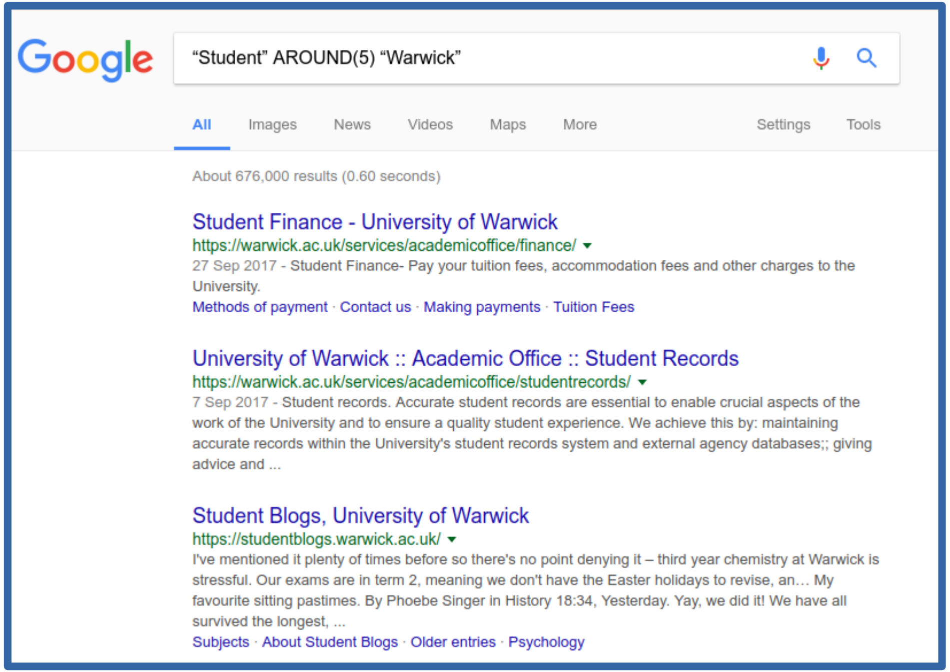
OTHER CHALLENGES IN INFORMATION RETRIEVAL

PROXIMITY QUERIES

- We may still want to do **more sophisticated queries, e.g. proximity queries**.
  - Search for two phrases, which occur within a maximum distance of K words between them.

PROXIMITY QUERIES

WARWICK



PROXIMITY QUERIES

- Query: **Student AROUND(5) Warwick**.
  - We can do this with positional indices.
  - With bigram indices we can't.

- The **algorithm for getting the intersection** here is more complex:
  - We need to get the intersection of "student" and "Warwick".
  - with the restriction that there has to be a maximum of 5 words in between.

PROXIMITY QUERIES: ALGORITHM

```
PositionalIntersect(p1, p2, k)
answer ← ⟨ ⟩
while p1 ≠ NIL and p2 ≠ NIL
do if docID(p1) = docID(p2)
     then l ← ⟨ ⟩
          pp1 ← positions(p1)
          pp2 ← positions(p2)
          while pp1 ≠ NIL
          do while pp2 ≠ NIL
             do if |pos(pp1) − pos(pp2)| ≤ k
                   then Add(l, pos(pp2))
                   else if pos(pp2) > pos(pp1)
                           then break
                pp2 ← next(pp2)
             while l ≠ ⟨ ⟩ and |l[0] − pos(pp1)| > k
             do Delete(l[0])
             for each ps ∈ l
             do Add(answer, ⟨docID(p1), pos(pp1), ps⟩)
             pp1 ← next(pp1)
          p1 ← next(p1)
          p2 ← next(p2)
     else if docID(p1) < docID(p2)
            then p1 ← next(p1)
            else p2 ← next(p2)
return answer
```

WILDCARD SEARCHES

- Query: **University** * **Warwick**.
- We want to look for **just one word** in between (the meaning of * is different from regex here).
- Positional indices can handle this, just as with **proximity queries where K=1**.

POSITIONAL INDEX SIZE

WARWICK

- Positional index uses **more space** than a binary index.
- However, positional indices are today's **standard approach to index documents**, given their **flexibility** for searching.

POSITIONAL INDEX SIZE

- Rules of thumb:
  - A **positional index is 2–4 as large** as a non-positional index.
  - Positional index size **35–50% of volume of original text.**
    - **Caveat:** all of this holds for “English-like” languages.

COMBINATION SCHEMES

WARWICK

- Positional indices and bigram indices **can be combined** to get the most of each approach:
  - For **frequent phrases** (“Michael Jackson”, “Britney Spears”, “The Who”) it is inefficient to keep on merging positional postings lists.

- For **very popular search queries**, you can also **cache** the results.

COMBINATION SCHEMES

- Williams et al. (2004) evaluated a **mixed indexing scheme**.
  - A typical web query mixture was **executed in ¼ of the time** of using just a positional index.
  - It required **26% more space** than having a positional index alone.

WARWICK

MORE POSSIBLE SEARCHES

- **Case sensitive search:** if we lowercase everything before indexing, we can't consider the case when searching.

- **Search page titles only:** need an additional flag to indicate that word position belongs to the title.

- **Search numeric ranges:** e.g. £50..£100. If numbers are just another string in our indices, we won't be able to search for this.

MORE CHALLENGES

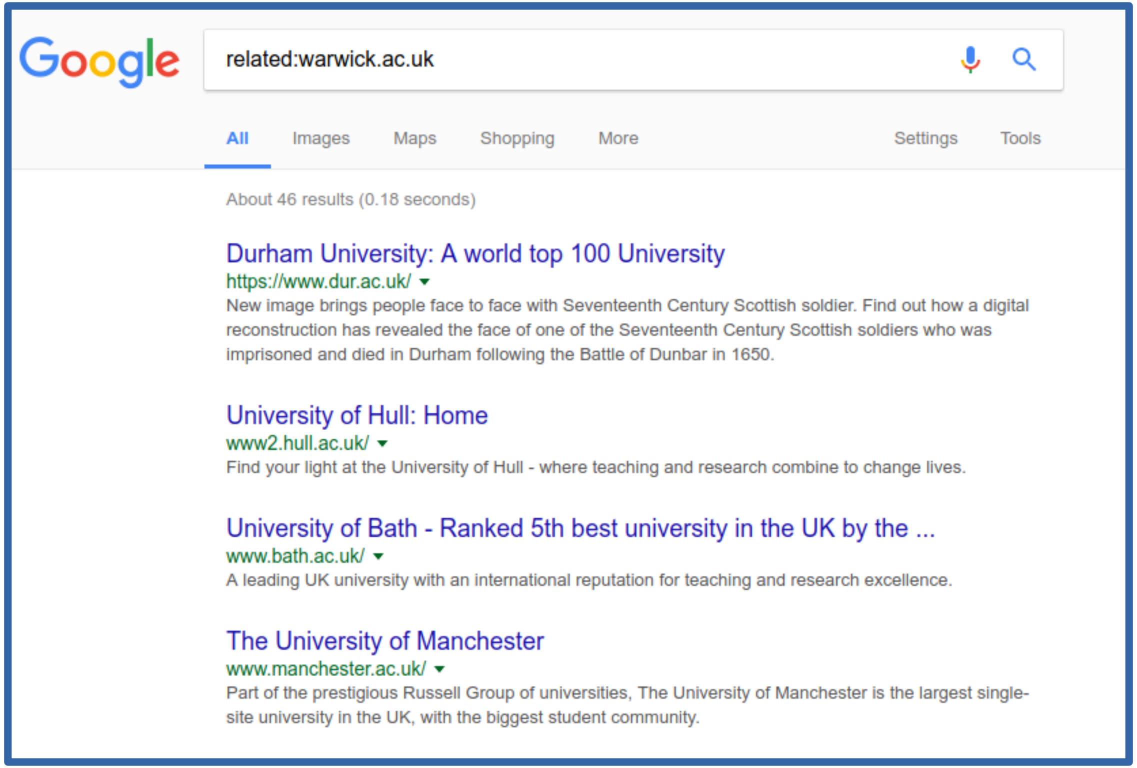
- For all these searches, the **challenge lies in implementing it efficiently**:
  - **Case: we could store original and lowercased words**, but that **increases the index size** substantially.
  - We could also have **separate indices for page titles and for numeric values**, but again, that's much more data to store.

MORE CHALLENGES

- With **Google**, you can even search for **related pages**:
  e.g. related:warwick.ac.uk

MORE CHALLENGES

WARWICK



MORE CHALLENGES

- With **Google**, you can even search for **related pages**:

  e.g. related:warwick.ac.uk

  returns a bunch of university websites

- They may also be **storing content similarity between websites**.
  - Again, that's **much more data for the index**.

MORE CHALLENGES

- So far, we **can retrieve documents that match a query**.

- That's fine, but we often get **many results**.
  - And **we want to rank them**.
  - **Weigh them based on relevance**, not just binary match.

- Ranking is an additional challenge in information retrieval.
  - Next lecture!

RESOURCES

- Apache Lucene (open source search software, Java):
  https://lucene.apache.org/

- Python wrapper for Lucene:
  http://lucene.apache.org/pylucene/

- Apache Nutch (web crawler, can be integrated with Lucene to build a search engine):
  http://nutch.apache.org/

ASSOCIATED READING

- Manning, C. D., Raghavan, P., & Schütze, H. (2008). Introduction to information retrieval (Vol. 1, No. 1, p. 496). Cambridge: Cambridge university press. **Chapters 1-2**. https://nlp.stanford.edu/IR-book/pdf/irbookonlinereading.pdf