CSE 103: Probability and statistics **Fall 2010**

Topic 4 — Randomized algorithms, III

4.1 Information retrieval

When you go to Google and enter a query, like

```
what are treatment options for pneumonia
```

or

```
new song by radiohead
```

you immediately get back a list of highly suitable pages. It's as if the search engine were instantaneously able to look through the tens of billions of pages on the web and find the relevant ones. How does it pull this off? The answer is, by a combination of clever preprocessing, statistics, hashing, and clustering.

In fact, these basic techniques apply not just to web search but to any system for *information retrieval*: answering unstructured queries about a large collection of documents.

4.1.1 Preprocessing

There are at five crucial preprocessing steps for a search engine.

1. Give each webpage a *reliability score*.

   Most of the "information" on the web is grossly unreliable: spam, ignorant ravings, unfounded conjectures, and idle gossip. But we'll see (a bit later in the course) that it is possible to assess the reliability or authoritativeness of individual webpages — by analyzing the statistics of linkage patterns.

2. Ignore *near-duplicates* of webpages.

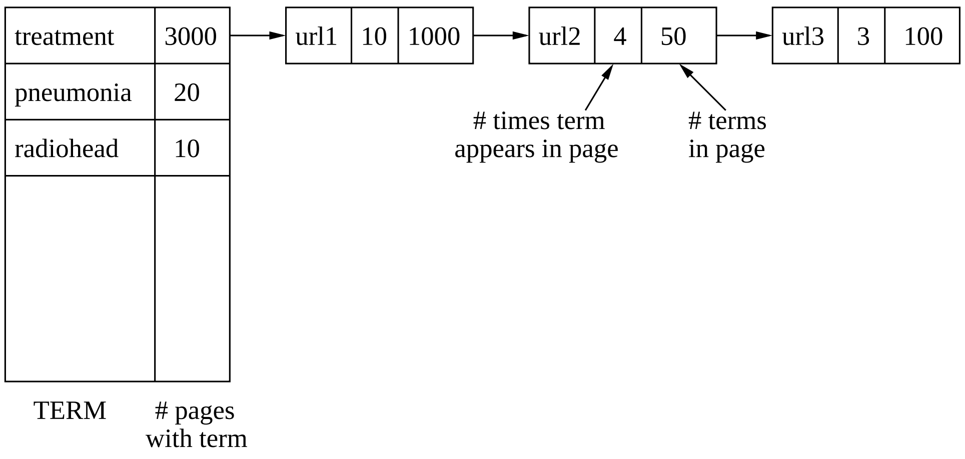
   More on this very shortly.

3. Discover the set of *terms*.

   A *term* is an individual word, or a sequence of words that should be considered together, such as `Katy Perry` or `Rage Against the Machine`. Terms can be discovered by analyzing co-occurrence patterns. If a sequence of k words keeps occurring together, they should be designated a term.

4. Create the *postings list*.

   This is a hash table in which each term is associated with a linked list of the pages containing it.



Each linked list is arranged in decreasing order of "priority" (some notion that takes into account reliability scores), and is usually truncated at a certain point.

5. Compute *term frequencies.*

   How common is each term?

4.1.2 Answering a query

The first step in answering a query is to order query terms by importance. The important terms are those that are uncommon. For instance, in

```
what are treatment options for pneumonia
```

the most important term is `pneumonia`, since this is the least common. Next comes `treatment`, and then `options`. The remaining words are so common that they can be ignored.

The next step is to look at the postings list, and get all pages containing the important query term(s). Then, assign each a score based on:

- the reliability of that page
- which of the important query terms it contains
- the number of times each query term appears on that page, divided by the length of the page

This yields an ordered list of webpages, starting with the "most relevant". But the list will typically contain way too many pages. So they need to be clustered, and the user can then be presented with one representative page from each cluster, with the option to select "more like this".

For instance, pages about `Rome` might be clustered into categories based on whether they refer to:

- Ancient Rome
- Modern Rome
- The TV series "Rome"
- and so on.

4.2 Detecting near-duplicates

Near-duplication is pervasive in the web: there are large numbers of distinct URLs which have exactly the same content but differ only in unimportant details like headers and footers. The user of a search engine would not be pleased if the answer to his query was a set of 10 near-identical pages! In order to remove this redundancy, we need to define a notion of *similarity* between documents.

4.2.1 The similarity between two documents

For any document—call it d—let the set of all words in d be denoted $C(d)$. For two documents d and d', we will measure their similarity by the function

$$S(d,d') \;=\; \frac{|C(d) \cap C(d')|}{|C(d) \cup C(d')|}.$$

If the two documents are truly identical, $S(d,d') = 1$. If they are almost-identical, $S(d,d')$ will be close to 1. And if they are completely different, with no words in common, then $S(d,d')$ will be zero. We'll consider d and d' to be near-duplicates if $S(d,d')$ is sufficiently close to 1.

Now, imagine a search engine that is going through a list of documents or webpages, and wants to eliminate near-duplicates. Here's an algorithm it could use:

- $\mathcal{D} = \emptyset$ (set of documents, initially empty)
- for each document d that appears:
  - if $S(d,d')$ is significantly smaller than 1 for all d' in $\mathcal{D}$: add d to $\mathcal{D}$

The final set of documents $\mathcal{D}$ will contain no near-duplicates. This is good, but the algorithm is very slow. Suppose for the sake of simplicity that there are n documents in total, each of length L. Then computing the similarity between two documents takes $O(L)$ time, and the algorithm is $O(n^2L)$. This quadratic dependence on n is prohibitive in web-scale applications, where n could easily be in the billions or tens of billions.

To get a faster algorithm, we once again resort to hashing.

4.2.2 An algorithm based on random permutations

We will encode each document by a single number. Here's how.

- Pick any encoding of words as numbers: for instance, any word is in any case stored as a binary number in the computer, and we can just use that number. Let $e(w)$ be the encoding of word w. Suppose these encodings are in the range $1,\ldots,M$.
- Let σ be a *random permutation* of $(1,2,\ldots,M)$. Thus for each i, $\sigma(i)$ is a number in the range 1 to M, and all the $\sigma(i)$ are different.
- Hash each document d to the single number

  $$f(d) = \min\{\sigma(e(w)) : w \in d\}.$$

  That is, first think of all the words in the document as numbers, then apply the random permutation to each of these numbers (to get a different set of numbers), and finally pick the smallest of these resulting numbers.

We will use the single number $f(d)$ in place of the entire document d! The rationale for doing this is captured in the following lemma, which says that near-duplicate documents are likely to be hashed to the same value.

Lemma 1. *Let d, d' be any two documents. If σ is a random permutation, then*

$$\Pr(f(d) = f(d')) \;=\; S(d, d').$$

Proof. For any word w, we will call $\sigma(e(w))$ its *value.*

Now, $f(d)$ and $f(d')$ will be equal if and only if the word in d with the smallest value is the same as the word in d' with the smallest value. This is the same as saying that the smallest value among words in $d \cup d'$ lies in $d \cap d'$. The probability of this is exactly

$$\frac{\#\text{ words in } d \cap d'}{\#\text{ words in } d \cup d'} \;=\; S(d, d').$$

Reason: σ is a random permutation, so each word in $d \cup d'$ is equally likely to be the one with the smallest value. □

Here's the revised algorithm.

- Create a boolean array `seen`$[1 \ldots M]$, initialized to `false`
- $\mathcal{D} = \emptyset$ (set of documents, initially empty)
- for each document d that appears:
  - if not `seen`$[f(d)]$: add d to $\mathcal{D}$ and set `seen`$[f(d)] =$ `true`

This time, the running time is $O(nL)$, just linear in n.