Lecture 12

12.1 Near duplicate documents[1]

Suppose we are designing a major search engine. We would like to avoid answering user queries with multiple copies of the same page. That is, there may be several pages with exactly the same text. These duplicates occur for a variety of reasons. Some are mirror sites, some are copies of common pages (such as Unix man pages), some are multiple spam advertisements, etc. Returning just one of the duplicates should be sufficient for the end user; returning all of them will clutter the response page, wasting valuable real estate and frustraing the user. How can we cope with duplicate pages?

Determining exact duplicates has a simple solution, based on hashing. Use the text of each page and an appropriate hash function to hash the text into a 64 bit signature. If two documents have the same signature, it is reasonable to assume that they share the same text. (Why? How often is this assumption wrong? Is it a terrible thing if the assumption turns out to be false?) By comparing signatures on the fly, we can avoid returning duplicates.

This solution works extremely well if we want to catch exact duplicates. What if, however, we want to capture the idea of "near duplicate" documents, or *similar* documents. For example, consider two mirror sites on the Web. It may be that the documents share the same text, except that the text corresponding to the links on the page are different, with each referring to the correct mirror site. In this case, the two pages will not yield the same signature, although again, we would not want to return both pages to the end user, because they are so similar. As another example, consider two copies of a newspaper article, one with a proper copyright notice added, and one without. We do not need to return both pages to the user. Again, hashing the document appears to be of no help. Finally, consider the case of advertisers who submit slightly modified versions of their ads over and over again, trying to get more or better spots on the response pages sent back to users. We want to stop their nefarious plans!

We will describe a scheme used to detect similar documents efficiently, using a hashing based scheme. Like the Bloom filter solution for password dictionaries, our solution is highly efficient in terms of space and time. The cost for this efficiency is accuracy; our algorithm will sometimes make mistakes, because it uses randomness.

12.2 Set resemblance

We describe a more general problem that will relate to our document similarity problem.

Consider two sets of numbers, A and B. For concreteness, we will assume that A and B are subsets of 64 bit numbers. We may define the *resemblance* of A and B as

$$\text{resemblance}(A,B) = R(A,B) = \frac{|A \cap B|}{|A \cup B|}.$$

The resemblance is a real number between 0 and 1. Intuitively, the resemblance accurately captures how close the two sets are. Sets and documents will be related, as we will see later.

[1]This lecture is based on the work of Andrei Broder, who developed these ideas, and convinced Altavista to use them! (The second feat may have been even more difficult than the first.)

How quickly can we determine the resemblance of two sets? If the sets are each of size n, the natural approach (compare each element to in A to each element in B) is $O(n^2)$. We can do better by sorting the sets. Still, these approaches are all rather slow, when we consider that we will have many sets to deal with and hence many pairs of sets to consider.

Instead we should ocnsider relaxing the problem. Suppose that we do not need an exact calculation of the resemblance $R(A,B)$. A reasonable estimate or approximation of the resemblance will suffice. Also, since we will be answering a variety of queries over a long period of time, it makes sense to consider algorithms that first do a *preprocessing* phase, in order to handle the queries much more quickly. That is, we will first do some work, preparing the appropriate data structures and data in a preprocessing phase. The advantage of doing all this work in advance will be that queries regarding resemblance can then be quickly answered.

Our estimation process will require a black box that does the following: it produces an effective *random permutation* on the set of 64 bit numbers. What do we mean by a random permutation? Let us consider just the case of four bit number, of which there are 16. Suppose we write each number on a card. Generating a random permutation is like shuffling this deck of 16 cards and looking at the order at which the numbers appear after this shuffling. For example, if we find the number 0011 on the first card, then our permutation maps the number 3 to the number 1. We write this as $\pi(3) = 1$, where π is a function that represents the permutation.

Suppose we have an efficient implemenation of random permutations, which we think of as a black box procedure. That is, when we invoke the black box procedure $BB(1,x)$ on a 64 bit number x, we get out $y = \pi_1(x)$ for some fixed, completely random permutation π_1. Similarly, if we invoke the black box $BB(2,x)$, we get out $\pi_2(x)$ for some different random permutation π_2. (In fact in practice we cannot achieve this black box, but we can get close enough that it is useful to think in these terms for analysis.)

Let us use the notation $\pi_1(A)$ to denote the set of elements obtained by computing $BB(1,x)$ for every x in A. Consider the following procedure: we compute the set $\pi_1(A)$ and $\pi_1(B)$, and record the *minimum* of each set. When does $\min\{\pi_1(A)\} = \min\{\pi_1(B)\}$? This happens only when there is some element x satisfying $\pi_1(x) = \min\{\pi_1(A)\} = \min\{\pi_1(B)\}$. In other words, the element x that is the minimum element in the set $A \cup B$ has to be the intersection of the sets $A \cap B$.

If π_1 is a random permutation, then every element in $A \cup B$ has equal probability of mapping to the minimum element after the permutation is applies. That is, for all x and y in $A \cup B$,

$$\mathbf{Pr}[\pi_1(x) = \min\{\pi_1(A \cup B)\}] = \mathbf{Pr}[\pi_1(y) = \min\{\pi_1(A \cup B)\}].$$

Thus, for the minimum of $\pi_1(A)$ and $\pi_1(B)$ to be the same, the minimum element must lie in $\pi_1(A \cap B)$ (see Figure 12.1). Hence

$$\mathbf{Pr}[\min\{\pi_1(A)\} = \min\{\pi_1(B)\}] = \frac{|A \cap B|}{|A \cup B|}.$$

But this is just the resemblance $R(A,B)$!

This gives us a way to estimate the resemblance. Instead of taking just one permutation, we take many– say 100. For each set A, we preprocess by computing $\min\{\pi_j(A)\}$ for $j = 1$ to 100, and store these values. To estimate the resemblance of two sets A and B, we count how often the minima are the same, and divide by 100. It is like each permutation gives us a coin flip, where the probability of a heads (a match) is exactly the resemblance $R(A,B)$ of the two sets.

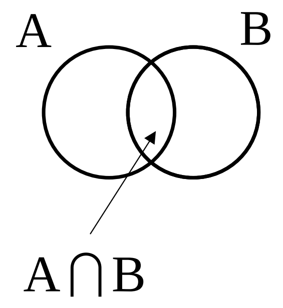

Figure 12.1: If the minimum element of $\pi_1(A)$ and $\pi_1(B)$ are the same, the minimum element must lie in $\pi_1(A \cap B)$.

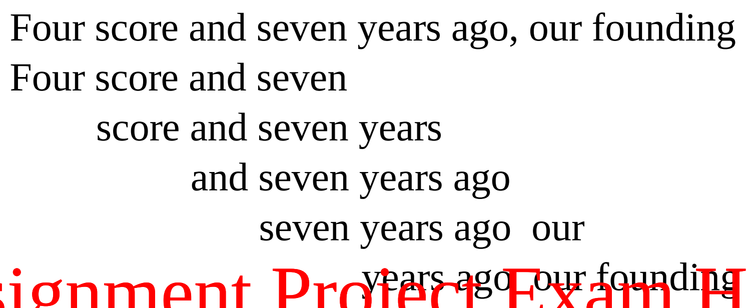

Figure 12.2: Shingling: the document is broken up into all segments of k consecutive words; each segment leads to a 64 bit hash value.

12.3 Turning Document Similarity into a Set Resemblance Problem

We now return to the original application. How do we turn document similarity into a set resemblance problem? The key idea is to hash pieces of the document– say every four consecutive words– into 64 bit numbers. This process has been called *shingling*, and each set of consecutive words is called a *shingle*. (See Figure 12.2.) Using hashing, the shingles give rise to the resulting numbers for the set resemblance problem, so that for each document D there is a set S_D. There are many possible variations and improvements possible. For example, one could modify the number of bits in a shingle or the method for shingling. Similarly, one could throw out all shingles that are not 0 mod 16, say, in order to reduce the number of shingles per document.

This approach obscures some important information in the document– such as the order paragraphs appear in, say. However, it seems reasonable to say that if the resulting sets have high resemblance, the documents are reasonably similar.

Once we have the shingles for the document, we associate a document *sketch* with each document. The sketch of a document S_D is a list of say 100 numbers: $(\min\{\pi_1(S_D)\}, \min\{\pi_2(S_D)\}, \min\{\pi_3(S_D)\}, \ldots, \min\{\pi_{100}(S_D)\})$.

Now we choose a threshold– for example, we might say that two documents are the similar if 90 out of the 100 entries in the sketch match. Now whenever a user queries the search engine, we check the sketches of the documents we wish to return. If two sketches share 90 entries, we only send one of them. (Alternatively, we could catch the duplicates on the crawling side– we check all the documents as we crawl the Web, and whenever two sketches share more than 90 entries, we assume the associated documents are similar, so that we only need to store one of them!)

Recall that our scheme uses random permutations. So, if we set our sketch threshold to 90 out of 100 entries, this does not guarantee that any pair of documents with high resemblance are caught. Also, some pairs of documents that do not have high resemblance may get marked as having high resemblance. How well does this scheme do?

We analyze how well the scheme does with the following argument. For each permutation π_i, the probability that two documents A and B have the same value in the ith position of the sketch is just the resemblance of the two documents $R(A,B) = r$. (Here the resemblance $R(A,B)$ of course refers to the resemblance of the sets of numbers obtained by shingling A and B.) Hence, the probability $p(r)$ that at least 90 out of the 100 entries in the sketch match is

$$p(r) = \sum_{k=90}^{100} \binom{100}{k} r^k (1-r)^{100-k}.$$

What does $p(r)$ look like as a function of r? The graph is shown in Figure 12.3. Notice that $p(r)$ stays very small until r approaches 0.9, and then quickly grows towards 1. This is exactly the property we want our scheme to have– if two documents are not similar, we will rarely mistake them for being similar, and if they are similar, we are likely to catch them!

For example, even if the resemblance is 0.8, we will only get 90 matches with probability less than 0.006! When the resemblance is only 0.5, the probability of having 90 entries in the sketch match falls to almost 10^{-18}! If documents are not alike, we will rarely mistake them as being similar.

If documents are alike, we will most likely catch them. If the resemblance is 0.95, the documents will have 90 or more entries in common in the sketch with probability greater than .988; if the resemblance is 0.96, the probability jumps to over .997.

We are dealing with a very large number of dcouments– most search engines currently index twenty-five to over one hundred million Web pages! So even though the probability of making a mistake is small, it will happen. The worst that happens, though, is that the search engine fails to index a few pages that it should, and it fails to catch a few duplicates that it should. These problems are not a big deal.

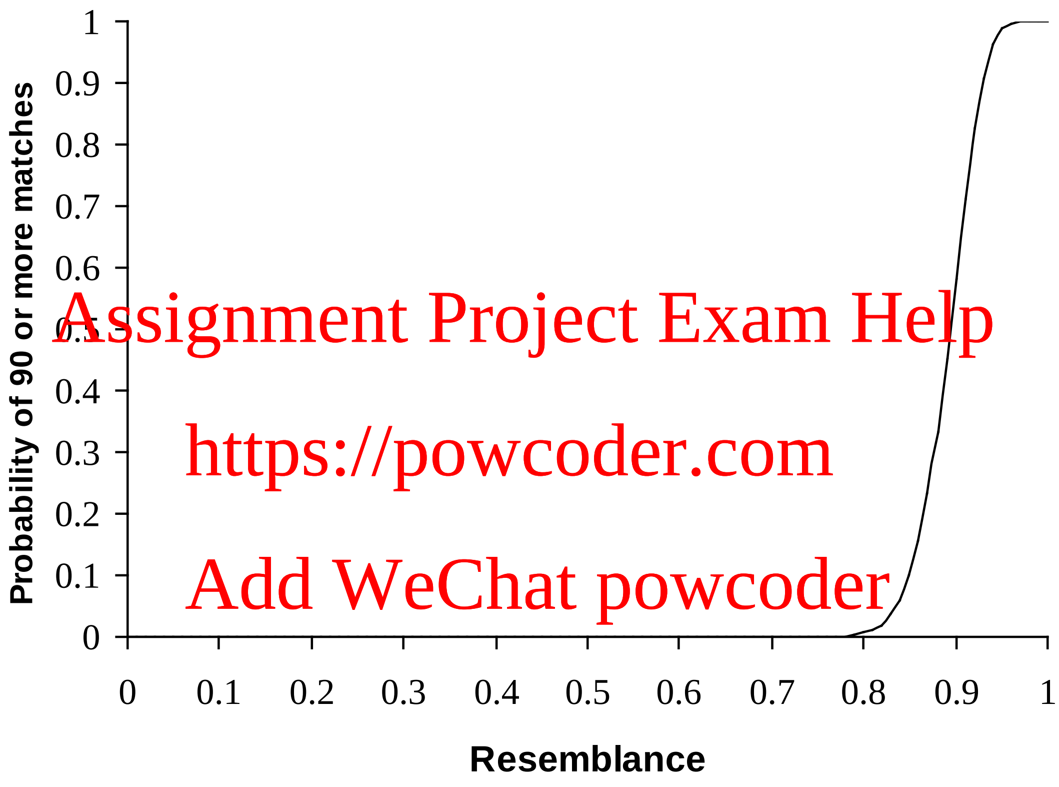

Figure 12.3: Making the threshold for document similarity 90 out of 100 matches in the sketch leads to the following graph relating resemblance to the probability two documents are considered similar. Notice the sharp change in behavior near where the resemblance is 0.90. Essentially, the procedure behaves like a low pass filter.