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.

2.0.1 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:

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

The final set of documents \( D \) 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.

2.0.2 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 \( \sigma \) 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. It is important that the same permutation \( \sigma \) is used for all the documents.

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

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

In practice, this algorithm is run not with the words in each document but with all sequences of \( k \) words (called “\( k \)-shingles”). For instance, the document

the quick brown fox jumped over the lazy dog

has the following 3-shingles: the quick brown, quick brown fox, brown fox jumped, fox jumped over, jumped over the, over the lazy, the lazy dog.