Finding Similar Items

A fundamental data-mining problem is to examine data for "similar" items. We shall take up applications in Section1.1, but an example would be looking at a collection of Web pages and finding near-duplicate pages. These pages could be plagiarisms, for example, or they could be mirrors that have almost the same content but differ in information about the host and about other mirrors.

We begin by phrasing the problem of similarity as one of finding sets with a relatively large intersection. We show how the problem of finding textually similar documents can be turned into such a set problem by the technique known as "shingling." Then, we introduce a technique called "minhashing," which compresses large sets in such a way that we can still deduce the similarity of the underlying sets from their compressed versions. Other techniques that work when the required degree of similarity is very high are cover ed in Section1.9.

Another important problem that arises when we search for similar items of any kind is that there may be far too many pairs of items to test each pair for their degree of similarity, even if computing the similarity of any one pair can be made very easy. That concern motivates a technique called "locality-sensitive hashing," for focusing our search on pairs that are most likely to be similar.

Finally, we explore notions of "similarity" that are not expressible as inter-section of sets. This study leads us to consider the theory of distance measures in arbitrary spaces. It also motivates a general framework for locality-sensitive hashing that applies for other definitions of "similarity."

1.1 Applications of Near-Neighbor Search

We shall focus initially on a particular notion of "similarity": the similarity of sets by looking at the relative size of their intersection. This notion of similarity is called "Jaccard similarity," and will be introduced in Section1.1.1. We then examine some of the uses of finding similar sets. These include finding textually similar documents and collaborative filtering by finding similar customers and similar products. In order to turn the problem of textual similarity of documents into one of set intersection , we use a technique called shingling, which is introduced in section 1.2

1.1.1 Jaccard Similarity of Sets

The **Jaccard similarity** of sets S and T is $|S \cap T|/|S \cup T|$, that is, the ratio of the size of the intersection of S and T to the size of their union. We shall denote the Jaccard similarity of S and T by SIM(S, T).

Example 1.1 In Fig.1.1 we see two sets S and T . There are three elements in their intersection and a total of eight elements that appear in S or T or both. Thus, SIM(S, T) = 3/8. 2

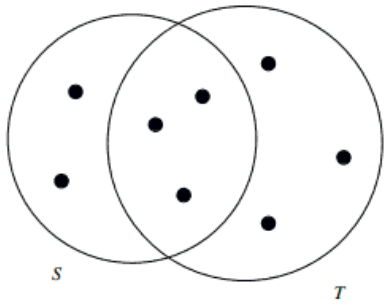

Figure 1.1: Two sets with Jaccard similarity 3/8

1.1.2 Similarity of Documents

An important class of problems that Jaccard similarity addresses well is that of finding textually similar documents in a large corpus such as the Web or a collection of news articles. We should understand that the aspect of similarity we are looking at here is character-level similarity, not "similar meaning," which requires us to examine the words in the documents and their uses. That problem is also interesting but is addressed by other techniques, which we hinted at in Section 1.3.1. However, textual similarity also has important uses. Many of these involve finding duplicates or near duplicates. First, let us observe that testing whether two documents are exact duplicates is easy; just compare the two documents character-by-character, and if they ever differ then they are not the same. However, in many applications, the documents are not identical, yet they share large portions of their text. Examples: Plagiarism, Mirror Pages, Articles from the Same Source etc.

1.2 Shingling of Documents

The most effective way to represent documents as sets, for the purpose of iden-tifying lexically similar documents is to construct from the document the set of short strings that appear within it. If we do so, then documents that share pieces as short as sentences or even phrases will have many common elements in their sets, even if those sentences appear in different orders in the two docu-ments. In this section, we introduce the simplest and most common approach, shingling, as well as an interesting variation.

1.2.1 k-Shingles

A document is a string of characters. Define a k-shingle for a document to be any substring of length k found within the document. Then, we may associate with each document the set of k-shingles that appear one or more times within that document.

Example 1.3 : Suppose our document D is the string abcdabd, and we pick k = 2. Then the set of 2-shingles for D is {ab, bc, cd, da, bd}.

Note that the substring ab appears twice within D, but appears only once as a shingle. A variation of shingling produces a bag, rather than a set, so each shingle would appear in the result as many times as it appears in the document. However, we shall not use bags of shingles here. 2

There are several options regarding how white space (blank, tab, newline, etc.) is treated. It probably makes sense to replace any sequence of one or more white-space characters by a single blank. That way, we distinguish shingles that cover two or more words from those that do not.

Example 1.4 : If we use k = 9, but eliminate whitespace altogether, then we would see some lexical similarity in the sentences "The plane was ready for touch down". and "The quarterback scored a touchdown". However, if we retain the blanks, then the first has shingles touch dow and ouch down, while the second has touchdown. If we eliminated the blanks, then both would have touchdown. 2

1.2.2 Choosing the Shingle Size

We can pick k to be any constant we like. However, if we pick k too small, then we would expect most sequences of k characters to appear in most documents. If so, then we could have documents whose shingle-sets had high Jaccard simi-larity, yet the documents had none of the same sentences or even phrases. As an extreme example, if we use k = 1, most Web pages will have most of the common characters and few other characters, so almost all Web pages will have high similarity.

How large k should be depends on how long typical documents are and how large the set of typical characters is. The important thing to remember is:

- k should be picked large enough that the probability of any given shingle appearing in any given document is low.

Thus, if our corpus of documents is emails, picking k = 5 should be fine. To see why, suppose that only letters and a general white-space character ap-pear in emails (although in practice, most of the printable ASCII characters can be expected to appear occasionally). If so, then there would be 27^5 = 14,348,907 possible shingles. Since the typical email is much smaller than 14 million characters long, we would expect k = 5 to work well, and indeed it does.

However, the calculation is a bit more subtle. Surely, more than 27 charac-ters appear in emails, However, all characters do not appear with equal proba-bility. Common letters and blanks dominate, while "z" and other letters that have high point-value in Scrabble are rare. Thus, even short emails will have many 5-shingles consisting of common letters, and the chances of unrelated emails sharing these common shingles is greater than would be implied by the calculation in the paragraph above. A good rule of thumb is to imagine that there are only 20 characters and estimate the number of k-shingles as 20^k . For large documents, such as research articles, choice k = 9 is considered safe.

1.2.3 Hashing Shingles

Instead of using substrings directly as shingles, we can pick a hash function that maps strings of length k to some number of buckets and treat the resulting bucket number as the shingle. The set representing a document is then the set of integers that are bucket numbers of one or more k-shingles that appear in the document. For instance, we could construct the set of 9-shingles for a document and then map each of those 9-shingles to a bucket number in the range 0 to $2^{32} - 1$. Thus, each shingle is represented by four bytes instead of nine. Not only has the data been compacted, but we can now manipulate (hashed) shingles by single-word machine operations.

Notice that we can differentiate documents better if we use 9-shingles and hash them down to four bytes than to use 4-shingles, even though the space used to represent a shingle is the same. The reason was touched upon in Section 1.2.2. If we use 4-shingles, most sequences of four bytes are unlikely or impossible to find in typical documents. Thus, the effective number of different shingles is much less than $2^{32} - 1$. If, as in Section 1.2.2, we assume only 20 characters are frequent in English text, then the number of different 4-shingles that are likely to occur is only $(20)^4$ = 160,000. However, if we use 9-shingles, there are many more than 2^{32} likely shingles. When

we hash them down to four bytes, we can expect almost any sequence of four bytes to be possible, as was discussed in Section 1.3.2.

1.2.4 Shingles Built from Words

An alternative form of shingle has proved effective for the problem of identifying similar news articles, mentioned in Section 1.1.2. The exploitable distinction for this problem is that the news articles are written in a rather different style than are other elements that typically appear on the page with the article. News articles, and most prose, have a lot of stop words (see Section 1.3.1), the most common words such as "and," "you," "to," and so on. In many applications, we want to ignore stop words, since they don't tell us anything useful about the article, such as its topic.

However, for the problem of finding similar news articles, it was found that defining a shingle to be a stop word followed by the next two words, regardless of whether or not they were stop words, formed a useful set of shingles. The advantage of this approach is that the news article would then contribute more shingles to the set representing the Web page than would the surrounding ele-ments. Recall that the goal of the exercise is to find pages that had the same articles, regardless of the surrounding elements. By biasing the set of shingles in favor of the article, pages with the same article and different surrounding material have higher Jaccard similarity than pages with the same surrounding material but with a different article.

Example 1.5 : An ad might have the simple text "Buy Sudzo." However, a news article with the same idea might read something like "**A** spokesperson **for the** Sudzo Corporation revealed today **that** studies **have** shown **it is** good **for** people **to** buy Sudzo products." Here, we have italicized all the likely stop words, although there is no set number of the most frequent words that should be considered stop words. The first three shingles made from a stop word and the next two following are:

A spokesperson for for the Sudzo
the Sudzo Corporation

There are nine shingles from the sentence, but none from the "ad." 2

1.3 Similarity-Preserving Summaries of Sets

Sets of shingles are large. Even if we hash them to four bytes each, the space needed to store a set is still roughly four times the space taken by the document. If we have millions of documents, it may well not be possible to store all the shingle-sets in main memory.[2]

Our goal in this section is to replace large sets by much smaller represen-tations called "signatures." The important property we need for signatures is that we can compare the signatures of two sets and estimate the Jaccard sim-ilarity of the underlying sets from the signatures alone. It is not possible that the signatures give the exact similarity of the sets they represent, but the esti-mates they provide are close, and the larger the signatures the more accurate the estimates. For example, if we replace the 200,000-byte hashed-shingle sets that derive from 50,000-byte documents by signatures of 1000 bytes, we can usually get within a few percent.

1.3.2 Minhashing

The signatures we desire to construct for sets are composed of the results of a large number of calculations, say several hundred, each of which is a "minhash" of the characteristic matrix. In this section, we shall learn how a

minhash is computed in principle, and in later sections we shall see how a good approxi-mation to the minhash is computed in practice.

To **minhash** a set represented by a column of the characteristic matrix, pick a permutation of the rows. The minhash value of any column is the number of the first row, in the permuted order, in which the column has a 1.

Example 1.7 : Let us suppose we pick the order of rows beadc for the matrix of Fig. 1.2. This permutation defines a minhash function h that maps sets to rows. Let us compute the minhash value of set S_1 according to h. The first column, which is the column for set S_1, has 0 in row b, so we proceed to row e, the second in the permuted order. There is again a 0 in the column for S_1 , so we proceed to row a, where we find a 1. Thus. $h(S_1) = a$.

Element	S_1	S_2	S_3	S_4
b	0	0	1	0
e	0	0	1	0
a	1	0	0	1
d	1	0	1	1
c	0	1	0	1

Figure 1.3: A permutation of the rows of Fig. 1.2

Although it is not physically possible to permute very large characteristic matrices, the minhash function h implicitly reorders the rows of the matrix of Fig. 1.2 so it becomes the matrix of Fig. 1.3. In this matrix, we can read off the values of h by scanning from the top until we come to a 1. Thus, we see that $h(S_2) = c$, $h(S_3) = b$, and $h(S_4) = a$. 2

1.3.3 Minhashing and Jaccard Similarity

There is a remarkable connection between minhashing and Jaccard similarity of the sets that are minhashed.

- The probability that the minhash function for a random permutation of rows produces the same value for two sets equals the Jaccard similarity of those sets.

To see why, we need to picture the columns for those two sets. If we restrict ourselves to the columns for sets S_1 and S_2, then rows can be divided into three classes:

1. Type X rows have 1 in both columns.

2. Type Y rows have 1 in one of the columns and 0 in the other.

3. Type Z rows have 0 in both columns.

Since the matrix is sparse, most rows are of type Z. However, it is the ratio of the numbers of type X and type Y rows that determine both $SIM(S_1 , S_2)$ and the probability that $h(S_1) = h(S_2)$. Let there be x rows of type X and y rows of type Y . Then $SIM(S_1, S_2) = x/(x + y)$. The reason is that x is the size of $S_1 \cap S_2$ and x + y is the size of $S_1 \cup S_2$.

Now, consider the probability that $h(S_1) = h(S_2)$. If we imagine the rows permuted randomly, and we proceed from the top, the probability that we shall meet a type X row before we meet a type Y row is $x/(x + y)$. But if the first row from the top other than type Z rows is a type X row, then surely $h(S_1) = h(S_2)$. On the other hand, if the first row other than a type Z row that we meet is a type Y row, then the set with a 1 gets that row as its minhash value. However the set with a 0 in that row surely gets some row further down the permuted list. Thus, we know $h(S_1) =6 h(S_2)$ if we first meet a type Y row. We conclude the probability that $h(S_1) = h(S_2)$ is $x/(x + y)$, which is also the Jaccard similarity of S_1 and S_2.

3.3.4 Minhash Signatures

Again think of a collection of sets represented by their characteristic matrix M . To represent sets, we pick at random some number n of permutations of the rows of M . Perhaps 100 permutations or several hundred permutations will do. Call the minhash functions determined by these permutations $h_1, h_2, \ldots, h_n$. From the column representing set S, construct the **minhash signature** for S, the vector $[h_1(S), h_2(S), \ldots, h_n(S)]$. We normally represent this list of hash-values as a column. Thus, we can form from matrix M a **signature matrix**, in which the ith column of M is replaced by the minhash signature for (the set of) the ith column.

Note that the signature matrix has the same number of columns as M but only n rows. Even if M is not represented explicitly, but in some compressed form suitable for a sparse matrix (e.g., by the locations of its 1's), it is normal for the signature matrix to be much smaller than M .

3.3.5 Computing Minhash Signatures

It is not feasible to permute a large characteristic matrix explicitly. Even picking a random permutation of millions or billions of rows is time-consuming, and the necessary sorting of the rows would take even more time. Thus, permuted matrices like that suggested by Fig. 1.3, while conceptually appealing, are not implementable.

Fortunately, it is possible to simulate the effect of a random permutation by a random hash function that maps row numbers to as many buckets as there are rows. A hash function that maps integers $0, 1, \ldots, k - 1$ to bucket numbers 0 through $k - 1$ typically will map some pairs of integers to the same bucket and leave other buckets unfilled. However, the difference is unimportant as long as k is large and there are not too many collisions. We can maintain the fiction that our hash function h "permutes" row r to position h(r) in the permuted order.

Thus, instead of picking n random permutations of rows, we pick n randomly chosen hash functions $h_1, h_2, \ldots, h_n$ on the rows. We construct the signature matrix by considering each row in their given order. Let SIG(i, c) be the element of the signature matrix for the ith hash function and column c. Initially, set SIG(i, c) to ∞ for all i and c. We handle row r by doing the following:

1. Compute $h_1(r), h_2(r), \ldots, h_n(r)$.

2. For each column c do the following:

   (a) If c has 0 in row r, do nothing.

   (b) However, if c has 1 in row r, then for each i = 1, 2, . . . , n set SIG(i, c) to the smaller of the current value of SIG(i, c) and $h_i(r)$.

Row	S_1	S_2	S_3	S_4	x + 1 mod 5	3x + 1 mod 5
0	1	0	0	1	1	1

1	0	0	1	0	2	4
2	0	1	0	1	3	2
3	1	0	1	1	4	0
4	0	0	1	0	0	3

Figure 1.4: Hash functions computed for the matrix of Fig. 1.2

Example 1.8 : Let us reconsider the characteristic matrix of Fig. 1.2, which we reproduce with some additional data as Fig. 1.4. We have replaced the letters naming the rows by integers 0 through 4. We have also chosen two hash functions: $h_1(x) = x+1 \bmod 5$ and $h_2(x) = 3x+1 \bmod 5$. The values of these two functions applied to the row numbers are given in the last two columns of Fig. 1.4. Notice that these simple hash functions are true permutations of the rows, but a true permutation is only possible because the number of rows, 5, is a prime. In general, there will be collisions, where two rows get the same hash value.

Now, let us simulate the algorithm for computing the signature matrix. Initially, this matrix consists of all ∞'s:

	S_1	S_2	S_3	S_4
h_1	∞	∞	∞	∞
h_2	∞	∞	∞	∞

First, we consider row 0 of Fig. 1.4. We see that the values of $h_1(0)$ and $h_2(0)$ are both 1. The row numbered 0 has 1's in the columns for sets S_1 and S_4, so only these columns of the signature matrix can change. As 1 is less than ∞, we do in fact change both values in the columns for S_1 and S_4. The current estimate of the signature matrix is thus:

	S_1	S_2	S_3	S_4
h_1	1	∞	∞	1
h_2	1	∞	∞	1

Now, we move to the row numbered 1 in Fig. 1.4. This row has 1 only in S_3, and its hash values are $h_1(1) = 2$ and $h_2(1) = 4$. Thus, we set SIG(1, 3) to 2 and SIG(2, 3) to 4. All other signature entries remain as they are because their columns have 0 in the row numbered 1. The new signature matrix:

	S_1	S_2	S_3	S_4
h1	1	∞	2	1
h2	1	∞	4	1

The row of Fig.1.4 numbered 2 has 1's in the columns for S_2 and S_4, and its hash values are $h_1(2) = 3$ and $h_2(2) = 2$. We could change the values in the signature for S_4, but the values in this column of the signature matrix, [1, 1], are each less than the corresponding hash values [3, 2]. However, since the column for S_2 still has ∞'s, we replace it by [3, 2], resulting in:

	S_1	S_2	S_3	S_4
h_1	1	3	2	1

h_2	1	2	4	1

Next comes the row numbered 3 in Fig.1.4. Here, all columns but S_2 have 1, and the hash values are $h_1(3) = 4$ and $h_2(3) = 0$. The value 4 for h_1 exceeds what is already in the signature matrix for all the columns, so we shall not change any values in the first row of the signature matrix. However, the value 0 for h_2 is less than what is already present, so we lower SIG(2, 1), SIG(2, 3) and SIG(2, 4) to 0. Note that we cannot lower SIG(2, 2) because the column for S_2 in Fig. 1.4 has 0 in the row we are currently considering. The resulting signature matrix:

	S1	S2	S3	S4
h_1	1	3	2	1
h2	0	2	0	0

Finally, consider the row of Fig. 1.4 numbered 4. $h_1(4) = 0$ and $h_2(4) = 3$. Since row 4 has 1 only in the column for S_3 , we only compare the current signature column for that set, [2, 0] with the hash values [0, 3]. Since $0 < 2$, we change SIG(1, 3) to 0, but since $3 > 0$ we do not change SIG(2, 3). The final signature matrix is:

	S1	S2	S3	S4
h_1	1	3	0	1
h_2	0	2	0	0

We can estimate the Jaccard similarities of the underlying sets from this signature matrix. Notice that columns 1 and 4 are identical, so we guess that $SIM(S_1 , S_4) = 1.0$. If we look at Fig. 1.4, we see that the true Jaccard similarity of S_1 and S_4 is 2/3. Remember that the fraction of rows that agree in the signature matrix is only an estimate of the true Jaccard similarity, and this example is much too small for the law of large numbers to assure that the estimates are close. For additional examples, the signature columns for S_1 and S_3 agree in half the rows (true similarity 1/4), while the signatures of S_1 and S_2 estimate 0 as their Jaccard similarity (the correct value). 2

1.4 Locality-Sensitive Hashing for Documents

Even though we can use minhashing to compress large documents into small signatures and preserve the expected similarity of any pair of documents, it still may be impossible to find the pairs with greatest similarity efficiently. The reason is that the number of pairs of documents may be too large, even if there are not too many documents.

Example 1.9 : Suppose we have a million documents, and we use signatures of length 250. Then we use 1000 bytes per document for the signatures, and the entire data fits in a gigabyte – less than a typical main memory of a laptop.

However, there are $2^{1,000,000}$ or half a trillion pairs of documents. If it takes a microsecond to compute the similarity of two signatures, then it takes almost six days to compute all the similarities on that laptop.

If our goal is to compute the similarity of every pair, there is nothing we can do to reduce the work, although parallelism can reduce the elapsed time. However, often we want only the most similar pairs or all pairs that are above some lower bound in similarity. If so, then we need to focus our attention only on pairs that are likely to be similar, without investigating every pair. There is a general theory of how to provide such focus, called **locality-sensitive hashing** (LSH) or **near-neighbor search**. In this section we shall consider a specific form of LSH, designed for the particular problem we have been studying: documents, represented by shingle-sets, then minhashed

to short signatures. In Section 1.6 we present the general theory of locality-sensitive hashing and a number of applications and related techniques.

3.4.1 LSH for Minhash Signatures

One general approach to LSH is to "hash" items several times, in such a way that similar items are more likely to be hashed to the same bucket than dissimilar items are. We then consider any pair that hashed to the same bucket for any of the hashings to be a **candidate pair**. We check only the candidate pairs for similarity. The hope is that most of the dissimilar pairs will never hash to the same bucket, and therefore will never be checked. Those dissimilar pairs that do hash to the same bucket are **false positives** ; we hope these will be only a small fraction of all pairs. We also hope that most of the truly similar pairs will hash to the same bucket under at least one of the hash functions. Those that do not are **false negatives**; we hope these will be only a small fraction of the truly similar pairs.

If we have minhash signatures for the items, an effective way to choose the hashings is to divide the signature matrix into b bands consisting of r rows each. For each band, there is a hash function that takes vectors of r integers (the portion of one column within that band) and hashes them to some large number of buckets. We can use the same hash function for all the bands, but we use a separate bucket array for each band, so columns with the same vector in different bands will not hash to the same bucket.

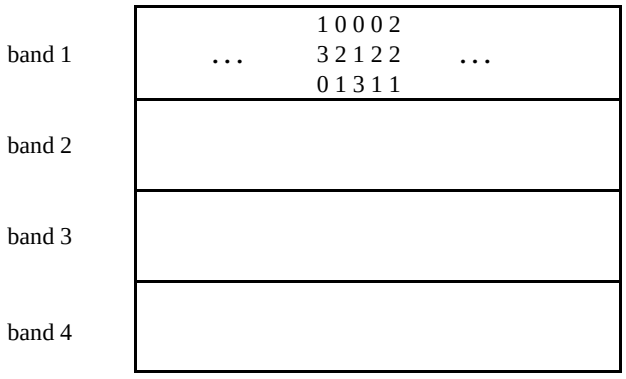

Figure 1.6: Dividing a signature matrix into four bands of three rows per band

Example 1.10 : Figure 1.6 shows part of a signature matrix of 12 rows divided into four bands of three rows each. The second and fourth of the explicitly shown columns each have the column vector [0, 2, 1] in the first band, so they will definitely hash to the same bucket in the hashing for the first band. Thus, regardless of what those columns look like in the other three bands, this pair of columns will be a candidate pair. It is possible that other columns, such as the first two shown explicitly, will also hash to the same bucket according to the hashing of the first band. However, since their column vectors are different, [1, 3, 0] and [0, 2, 1], and there are many buckets for each hashing, we expect the chances of an accidental collision to be very small. We shall normally assume that two vectors hash to the same bucket if and only if they are identical.

Two columns that do not agree in band 1 have three other chances to become a candidate pair; they might be identical in any one of these other bands. However, observe that the more similar two columns are, the more likely it is that they will be identical in some band. Thus, intuitively the banding strategy makes similar columns much more likely to be candidate pairs than dissimilar pairs.

1.4.2 Analysis of the Banding Technique

Suppose we use b bands of r rows each, and suppose that a particular pair of documents have Jaccard similarity s. Recall from Section 1.3.3 that the prob-ability the minhash signatures for these documents agree in any one particular row of the signature matrix is s. We can calculate the probability that these documents (or rather their signatures) become a candidate pair as follows:

1. The probability that the signatures agree in all rows of one particular band is s^r .

2. The probability that the signatures do not agree in at least one row of a particular band is $1 - s^r$.

3. The probability that the signatures do not agree in all rows of any of the bands is $(1 - s^r)^b$.

4. The probability that the signatures agree in all the rows of at least one band, and therefore become a candidate pair, is $1 - (1 - s^r)^b$.

It may not be obvious, but regardless of the chosen constants b and r, this function has the form of an **S-curve**, as suggested in Fig. 1.7. The **threshold**, that is, the value of similarity s at which the probability of becoming a candidate is 1/2, is a function of b and r. The threshold is roughly where the rise is the steepest, and for large b and r there we find that pairs with similarity above the threshold are very likely to become candidates, while those below the threshold are unlikely to become candidates – exactly the situation we want. An approximation to the threshold is $(1/b)^{1/r}$. For example, if b = 16 and r = 4, then the threshold is approximately at s = 1/2, since the 4th root of 1/16 is 1/2.

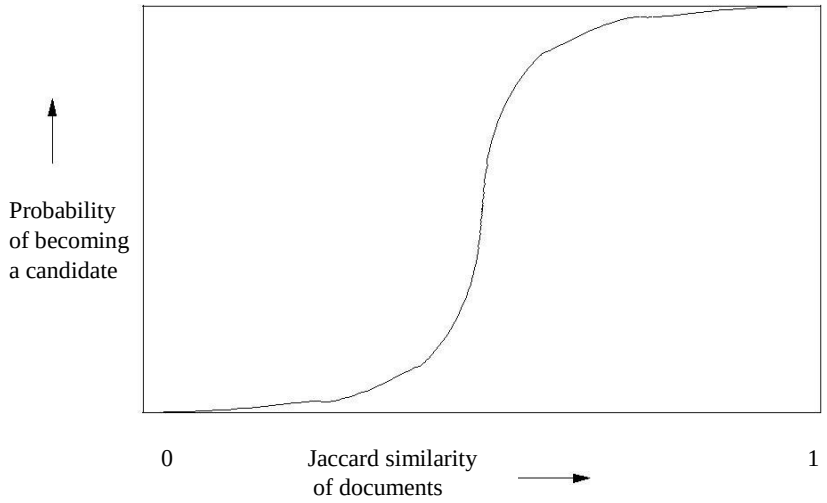

Figure 1.7: The S-curve

Example 1.11 : Let us consider the case b = 20 and r = 5. That is, we suppose we have signatures of length 100, divided into twenty bands of five rows each. Figure 1.8 tabulates some of the values of the function $1 - (1 - s^5)^{20}$. Notice that the threshold, the value of s at which the curve has risen halfway, is just slightly more than 0.5. Also notice that the curve is not exactly the ideal step function that jumps from 0 to 1 at the threshold, but the slope of the

curve in the middle is significant. For example, it rises by more than 0.6 going from s = 0.4 to s = 0.6, so the slope in the middle is greater than 3.

s	$1 - (1 - s^r)^b$
.2	.006
.3	.047
.4	.186
.5	.470
.6	.802
.7	.975
.8	.9996

Figure 1.8: Values of the S-curve for b = 20 and r = 5

For example, at s = 0.8, $1 - (0.8)^5$ is about 0.672. If you raise this number to the 20th power, you get about 0.00035. Subtracting this fraction from 1 yields 0.99965. That is, if we consider two documents with 80% similarity, then in any one band, they have only about a 33% chance of agreeing in all five rows and thus becoming a candidate pair. However, there are 20 bands and thus 20 chances to become a candidate. Only roughly one in 3000 pairs that are as high as 80% similar will fail to become a candidate pair and thus be a false negative.

1.4.3 Combining the Techniques

We can now give an approach to finding the set of candidate pairs for similar documents and then discovering the truly similar documents among them. It must be emphasized that this approach can produce false negatives – pairs of similar documents that are not identified as such because they never become a candidate pair. There will also be false positives – candidate pairs that are evaluated, but are found not to be sufficiently similar.

1. Pick a value of k and construct from each document the set of k-shingles. Optionally, hash the k-shingles to shorter bucket numbers.

2. Sort the document-shingle pairs to order them by shingle.

3. Pick a length n for the minhash signatures. Feed the sorted list to the algorithm of Section 1.3.5 to compute the minhash signatures for all the documents.

4. Choose a threshold t that defines how similar documents have to be in order for them to be regarded as a desired "similar pair." Pick a number

   of bands b and a number of rows r such that br = n, and the threshold t is approximately $(1/b)^{1/r}$. If avoidance of false negatives is important, you may wish to select b and r to produce a threshold lower than t; if speed is important and you wish to limit false positives, select b and r to produce a higher threshold.

5. Construct candidate pairs by applying the LSH technique of Section 1.4.1.

6. Examine each candidate pair's signatures and determine whether the frac-tion of components in which they agree is at least t.

7. Optionally, if the signatures are sufficiently similar, go to the documents themselves and check that they are

truly similar, rather than documents that, by luck, had similar signatures.

1.5 Distance Measures

1.5.1 Definition of a Distance Measure

Suppose we have a set of points, called a **space**. A **distance measure** on this space is a function d(x, y) that takes two points in the space as arguments and produces a real number, and satisfies the following axioms:

1. $d(x, y) \geq 0$ (no negative distances).
2. $d(x, y) = 0$ if and only if $x = y$ (distances are positive, except for the distance from a point to itself).
3. $d(x, y) = d(y, x)$ (distance is symmetric).
4. $d(x, y) \leq d(x, z) + d(z, y)$ (the **triangle inequality**).

The triangle inequality is the most complex condition. It says, intuitively, that to travel from x to y, we cannot obtain any benefit if we are forced to travel via some particular third point z. The triangle-inequality axiom is what makes all distance measures behave as if distance describes the length of a shortest path from one point to another.

1.5.5 Edit Distance

This distance makes sense when points are strings. The distance between two strings x = x1x2 ・ ・ ・ xn and y = y1y2 ・ ・ ・ ym is the smallest number of insertions and deletions of single characters that will convert x to y.

Example 1.14 : The edit distance between the strings x = abcde and y = acfdeg is 3. To convert x to y:

1. Delete b.

2. Insert f after c.

3. Insert g after e.

No sequence of fewer than three insertions and/or deletions will convert x to y.

Thus, d(x, y) = 3.

Notice that to satisfy the second axiom, we have to treat vectors that are multiples of one another, e.g. [1, 2] and [3, 6], as the same direction, which they are. If we regarded these as different vectors, we would give them distance 0 and thus violate the condition that only d(x, x) is 0.

1.5.3 Jaccard Distance

As mentioned at the beginning of the section, we define the **Jaccard distance** of sets by d(x, y) = 1 − SIM(x, y). That is, the Jaccard distance is 1 minus the ratio of the sizes of the intersection and union of sets x and y. We must verify that this function is a distance measure.

1. d(x, y) is nonnegative because the size of the intersection cannot exceed the size of the union.

2. d(x, y) = 0 if x = y, because x ∪ x = x ∩ x = x. However, if x 6= y, then the size of x ∩ y is strictly less than the size of x ∪ y, so d(x, y) is strictly positive.

3. d(x, y) = d(y, x) because both union and intersection are symmetric; i.e., x ∪ y = y ∪ x and x ∩ y = y ∩ x.

4. For the triangle inequality, recall from Section 1.3.3 that SIM(x, y) is the probability a random minhash function maps x and y to the same value. Thus, the Jaccard distance d(x, y) is the probability that a random min-hash function **does not** send x and y to the same value. We can therefore translate the condition d(x, y) ≤ d(x, z) + d(z, y) to the statement that if h is a random minhash function, then the probability that h(x) =6 h(y) is no greater than the sum of the probability that h(x) =6 h(z) and the probability that h(z) =6 h(y). However, this statement is true because whenever h(x) =6 h(y), at least one of h(x) and h(y) must be different from h(z). They could not both be h(z), because then h(x) and h(y) would be the same.

1.10 Summary of The Chapter 1

- **Jaccard Similarity**: The Jaccard similarity of sets is the ratio of the size of the intersection of the sets to the size of the union. This measure of similarity is suitable for many applications, including textual similarity of documents and similarity of buying habits of customers.

- **Shingling**: A k-shingle is any k characters that appear consecutively in a document. If we represent a document by its set of k-shingles, then the Jaccard similarity of the shingle sets measures the textual similarity of documents. Sometimes, it is useful to hash shingles to bit strings of shorter length, and use sets of hash values to represent documents.

- **Minhashing**: A minhash function on sets is based on a permutation of the universal set. Given any such permutation, the minhash value for a set is that element of the set that appears first in the permuted order.

- **Minhash Signatures** : We may represent sets by picking some list of per-mutations and computing for each set its minhash signature, which is the sequence of minhash values obtained by applying each permutation on the list to that set. Given two sets, the expected fraction of the permutations that will yield the same minhash value is exactly the Jaccard similarity of the sets.

- **Efficient Minhashing**: Since it is not really possible to generate random permutations, it is normal to simulate a permutation by picking a random hash function and taking the minhash value for a set to be the least hash value of any of the set's members.

- **Locality-Sensitive Hashing for Signatures** : This technique allows us to avoid computing the similarity of every pair of sets or their minhash sig-natures. If we are given signatures for the sets, we may divide them into bands, and only measure the similarity of a pair of sets if they are identi-cal in at least one band. By choosing

the size of bands appropriately, we can eliminate from consideration most of the pairs that do not meet our threshold of similarity.

✦ **Distance Measures** : A distance measure is a function on pairs of points in a space that satisfy certain axioms. The distance between two points is 0 if the points are the same, but greater than 0 if the points are diff erent. The distance is symmetric; it does not matter in which order we consider the two points. A distance measure must satisfy the triangle inequality: the distance between two points is never more than the sum of the distances between those points and some third point.

✦ **Jaccard Distance**: One minus the Jaccard similarity is a distance measure, called the Jaccard distance.

✦ **Edit Distance**: This distance measure applies to a space of strings, and is the number of insertions and/or deletions needed to convert one string into the other. The edit distance can also be computed as the sum of the lengths of the strings minus twice the length of the longest common subsequence of the strings.

1.11 References for Chapter 1

The technique we called shingling is attributed to [10]. The use in the manner we discussed here is from [2]. Minhashing comes from [3]. The original works on locality-sensitive hashing were [9] and [7]. [1] is a useful summary of ideas in this field.

[4] Introduces the idea of using random-hyperplanes to summarize items in a way that respects the cosine distance. [8] suggests that random hyperplanes plus LSH can be more accurate at detecting similar documents than minhashing plus LSH.

Techniques for summarizing points in a Euclidean space are covered in [6]. [11] presented the shingling technique based on stop words.

The length and prefix-based indexing schemes for high-similarity matching comes from [5]. The technique involving suffix length is from [12].

1. A. Andoni and P. Indyk, “Near-optimal hashing algorithms for approxi-mate nearest neighbor in high dimensions,” **Comm. ACM** 51:1, pp. 117– 122, 2008.

2. A.Z. Broder, “On the resemblance and containment of documents,” **Proc. Compression and Complexity of Sequences**, pp. 21–29, Positano Italy, 1997.

3. A.Z. Broder, M. Charikar, A.M. Frieze, and M. Mitzenmacher, “Min-wise independent permutations,” **ACM Symposium on Theory of Computing**, pp. 327–336, 1998.

4. M.S. Charikar, “Similarity estimation techniques from rounding algo-rithms,” **ACM Symposium on Theory of Computing**, pp. 380–388, 2002.

5. S. Chaudhuri, V. Ganti, and R. Kaushik, “A primitive operator for sim-ilarity joins in data cleaning,” **Proc. Intl. Conf. on Data Engineering**, 2006.

6. M. Datar, N. Immorlica, P. Indyk, and V.S. Mirrokni, “Locality-sensitive hashing scheme based on p-stable distributions,” **Symposium on Compu-tational Geometry** pp. 253–262, 2004.

7. A. Gionis, P. Indyk, and R. Motwani, “Similarity search in high dimen-sions via hashing,” **Proc. Intl. Conf. on Very Large Databases**, pp. 518– 529, 1999.

8. M. Henzinger, “Finding near-duplicate web pages: a large-scale evaluation of algorithms,” **Proc. 29th SIGIR Conf.**, pp. 284–291, 2006.

9. P. Indyk and R. Motwani. “Approximate nearest neighbor: towards re-moving the curse of dimensionality,” **ACM Symposium on Theory of Com-puting**, pp. 604–613, 1998.

10. U. Manber, “Finding similar files in a large file system,” **Proc. USENIX Conference**, pp. 1–10, 1994.

11. M. Theobald, J. Siddharth, and A. Paepcke, “SpotSigs: robust and effi-cient near duplicate detection in large web collections,” **31st Annual ACM SIGIR Conference**, July, 2008, Singapore.

12. C. Xiao, W. Wang, X. Lin, and J.X. Yu, “Efficient similarity joins for near duplicate detection,” **Proc. WWW Conference**, pp. 131-140, 2008.