Chapter 3

Finding Similar Items

A fundamental data-mining problem is to examine data for "similar" items. We shall take up applications in Section 3.1, 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 covered in Section 3.9.

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 intersection 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."

3.1 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 Section 3.1.1. 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 3.2.

3.1.1 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 $\text{SIM}(S, T)$.

Example 3.1 : In Fig. 3.1 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, $\text{SIM}(S, T) = 3/8$. □

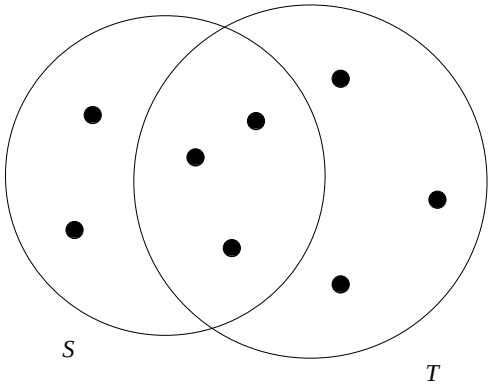

Figure 3.1: Two sets with Jaccard similarity 3/8

3.1.2 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. Here are some examples:

Plagiarism

Finding plagiarized documents tests our ability to find textual similarity. The plagiarizer may extract only some parts of a document for his own. He may alter a few words and may alter the order in which sentences of the original appear. Yet the resulting document may still contain 50% or more of the original. No simple process of comparing documents character by character will detect a sophisticated plagiarism.

Mirror Pages

It is common for important or popular Web sites to be duplicated at a number of hosts, in order to share the load. The pages of these *mirror* sites will be quite similar, but are rarely identical. For instance, they might each contain information associated with their particular host, and they might each have links to the other mirror sites but not to themselves. A related phenomenon is the appropriation of pages from one class to another. These pages might include class notes, assignments, and lecture slides. Similar pages might change the name of the course, year, and make small changes from year to year. It is important to be able to detect similar pages of these kinds, because search engines produce better results if they avoid showing two pages that are nearly identical within the first page of results.

Articles from the Same Source

It is common for one reporter to write a news article that gets distributed, say through the Associated Press, to many newspapers, which then publish the article on their Web sites. Each newspaper changes the article somewhat. They may cut out paragraphs, or even add material of their own. They most likely will surround the article by their own logo, ads, and links to other articles at their site. However, the core of each newspaper's page will be the original article. News aggregators, such as Google News, try to find all versions of such an article, in order to show only one, and that task requires finding when two Web pages are textually similar, although not identical.[1]

3.1.3 Collaborative Filtering as a Similar-Sets Problem

Another class of applications where similarity of sets is very important is called *collaborative filtering*, a process whereby we recommend to users items that were liked by other users who have exhibited similar tastes. We shall investigate collaborative filtering in detail in Section 9.3, but for the moment let us see some common examples.

[1] News aggregation also involves finding articles that are about the same topic, even though not textually similar. This problem too can yield to a similarity search, but it requires techniques other than Jaccard similarity of sets.

On-Line Purchases

Amazon.com has millions of customers and sells millions of items. Its database records which items have been bought by which customers. We can say two customers are similar if their sets of purchased items have a high Jaccard similarity. Likewise, two items that have sets of purchasers with high Jaccard similarity will be deemed similar. Note that, while we might expect mirror sites to have Jaccard similarity above 90%, it is unlikely that any two customers have Jaccard similarity that high (unless they have purchased only one item). Even a Jaccard similarity like 20% might be unusual enough to identify customers with similar tastes. The same observation holds for items; Jaccard similarities need not be very high to be significant.

Collaborative filtering requires several tools, in addition to finding similar customers or items, as we discuss in Chapter 9. For example, two Amazon customers who like science-fiction might each buy many science-fiction books, but only a few of these will be in common. However, by combining similarity-finding with clustering (Chapter 7), we might be able to discover that science-fiction books are mutually similar and put them in one group. Then, we can get a more powerful notion of customer-similarity by asking whether they made purchases within many of the same groups.

Movie Ratings

NetFlix records which movies each of its customers rented, and also the ratings assigned to those movies by the customers. We can see movies as similar if they were rented or rated highly by many of the same customers, and see customers as similar if they rented or rated highly many of the same movies. The same observations that we made for Amazon above apply in this situation: similarities need not be high to be significant, and clustering movies by genre will make things easier.

When our data consists of ratings rather than binary decisions (bought/did not buy or liked/disliked), we cannot rely simply on sets as representations of customers or items. Some options are:

1. Ignore low-rated customer/movie pairs; that is, treat these events as if the customer never watched the movie.

2. When comparing customers, imagine two set elements for each movie, "liked" and "hated." If a customer rated a movie highly, put the "liked" for that movie in the customer's set. If they gave a low rating to a movie, put "hated" for that movie in their set. Then, we can look for high Jaccard similarity among these sets. We can do a similar trick when comparing movies.

3. If ratings are 1-to-5-stars, put a movie in a customer's set n times if they rated the movie n-stars. Then, use *Jaccard similarity for bags* when measuring the similarity of customers. The Jaccard similarity for bags

B and C is defined by counting an element n times in the intersection if n is the minimum of the number of times the element appears in B and C. In the union, we count the element the sum of the number of times it appears in B and in C.[2]

Example 3.2: The bag-similarity of bags $\{a, a, a, b\}$ and $\{a, a, b, b, c\}$ is 1/3. The intersection counts a twice and b once, so its size is 3. The size of the union of two bags is always the sum of the sizes of the two bags, or 9 in this case. Since the highest possible Jaccard similarity for bags is 1/2, the score of 1/3 indicates the two bags are quite similar, as should be apparent from an examination of their contents. □

3.1.4 Exercises for Section 3.1

Exercise 3.1.1: Compute the Jaccard similarities of each pair of the following three sets: $\{1, 2, 3, 4\}$, $\{2, 3, 5, 7\}$, and $\{2, 4, 6\}$.

Exercise 3.1.2: Compute the Jaccard bag similarity of each pair of the following three bags: $\{1, 1, 1, 2\}$, $\{1, 1, 2, 2, 3\}$, and $\{1, 2, 3, 4\}$.

!! **Exercise 3.1.3:** Suppose we have a universal set U of n elements, and we choose two subsets S and T at random, each with m of the n elements. What is the expected value of the Jaccard similarity of S and T?

3.2 Shingling of Documents

The most effective way to represent documents as sets, for the purpose of identifying 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 documents. In this section, we introduce the simplest and most common approach, shingling, as well as an interesting variation.

3.2.1 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

[2] Although the union for bags is normally (e.g., in the SQL standard) defined to have the sum of the number of copies in the two bags, this definition causes some inconsistency with the Jaccard similarity for sets. Under this definition of bag union, the maximum Jaccard similarity is 1/2, not 1, since the union of a set with itself has twice as many elements as the intersection of the same set with itself. If we prefer to have the Jaccard similarity of a set with itself be 1, we can redefine the union of bags to have each element appear the maximum number of times it appears in either of the two bags. This change does not simply double the similarity in each case, but it also gives a reasonable measure of bag similarity.

with each document the set of k-shingles that appear one or more times within that document.

Example 3.3: Suppose our document D is the string `abcdabd`, and we pick $k = 2$. Then the set of 2-shingles for D is $\{\texttt{ab}, \texttt{bc}, \texttt{cd}, \texttt{da}, \texttt{bd}\}$.

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. □

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 3.4: 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`. □

3.2.2 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 similarity, 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 appear 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 characters appear in emails, However, all characters do not appear with equal probability. 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.

3.2.3 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 3.2.2. 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 3.2.2, 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.

3.2.4 Shingles Built from Words

An alternative form of shingle has proved effective for the problem of identifying similar news articles, mentioned in Section 3.1.2. 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 3.5: 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." □

3.2.5 Exercises for Section 3.2

Exercise 3.2.1: What are the first ten 3-shingles in the first sentence of Section 3.2?

Exercise 3.2.2: If we use the stop-word-based shingles of Section 3.2.4, and we take the stop words to be all the words of three or fewer letters, then what are the shingles in the first sentence of Section 3.2?

Exercise 3.2.3: What is the largest number of k-shingles a document of n bytes can have? You may assume that the size of the alphabet is large enough that the number of possible strings of length k is at least as n.

3.3 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.[3]

Our goal in this section is to replace large sets by much smaller representations called "signatures." The important property we need for signatures is that we can compare the signatures of two sets and estimate the Jaccard similarity of the underlying sets from the signatures alone. It is not possible that

[3]There is another serious concern: even if the sets fit in main memory, the number of pairs may be too great for us to evaluate the similarity of each pair. We take up the solution to this problem in Section 3.4.

the signatures give the exact similarity of the sets they represent, but the estimates 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.

3.3.1 Matrix Representation of Sets

Before explaining how it is possible to construct small signatures from large sets, it is helpful to visualize a collection of sets as their *characteristic matrix*. The columns of the matrix correspond to the sets, and the rows correspond to elements of the universal set from which elements of the sets are drawn. There is a 1 in row r and column c if the element for row r is a member of the set for column c. Otherwise the value in position (r, c) is 0.

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

Figure 3.2: A matrix representing four sets

Example 3.6 : In Fig. 3.2 is an example of a matrix representing sets chosen from the universal set $\{a, b, c, d, e\}$. Here, $S_1 = \{a, d\}$, $S_2 = \{c\}$, $S_3 = \{b, d, e\}$, and $S_4 = \{a, c, d\}$. The top row and leftmost columns are not part of the matrix, but are present only to remind us what the rows and columns represent. □

It is important to remember that the characteristic matrix is unlikely to be the way the data is stored, but it is useful as a way to visualize the data. For one reason not to store data as a matrix, these matrices are almost always *sparse* (they have many more 0's than 1's) in practice. It saves space to represent a sparse matrix of 0's and 1's by the positions in which the 1's appear. For another reason, the data is usually stored in some other format for other purposes.

As an example, if rows are products, and columns are customers, represented by the set of products they bought, then this data would really appear in a database table of purchases. A tuple in this table would list the item, the purchaser, and probably other details about the purchase, such as the date and the credit card used.

3.3.2 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 approximation 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 3.7: Let us suppose we pick the order of rows *beadc* for the matrix of Fig. 3.2. 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 3.3: A permutation of the rows of Fig. 3.2

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. 3.2 so it becomes the matrix of Fig. 3.3. 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 $h(S_4) = a$. □

3.3.3 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 $\text{SIM}(S_1, S_2)$ 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 $\text{SIM}(S_1, S_2) = x/(x + y)$. 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 $h(S_1) \neq h(S_2)$ 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. 3.3, 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 $\text{SIG}(i, c)$ be the element of the signature matrix for the ith hash function and column c. Initially, set $\text{SIG}(i, c)$ 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 $i = 1, 2, \ldots, n$ set $\text{SIG}(i, c)$ to the smaller of the current value of $\text{SIG}(i, c)$ 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 3.4: Hash functions computed for the matrix of Fig. 3.2

Example 3.8: Let us reconsider the characteristic matrix of Fig. 3.2, which we reproduce with some additional data as Fig. 3.4. We have replaced the letters naming the rows by integers 0 through 4. We have also chosen two hash functions: $h_1(x) = x+1 \mod 5$ and $h_2(x) = 3x+1 \mod 5$. The values of these two functions applied to the row numbers are given in the last two columns of Fig. 3.4. 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. 3.4. 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. 3.4. 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 $\text{SIG}(1,3)$ to 2 and $\text{SIG}(2,3)$ 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
h_1	1	∞	2	1
h_2	1	∞	4	1

The row of Fig. 3.4 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. 3.4. 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 $\text{SIG}(2,1)$, $\text{SIG}(2,3)$ and $\text{SIG}(2,4)$ to 0. Note that we cannot lower $\text{SIG}(2,2)$ because the column for S_2 in Fig. 3.4 has 0 in the row we are currently considering. The resulting signature matrix:

	S_1	S_2	S_3	S_4
h_1	1	3	2	1
h_2	0	2	0	0

Finally, consider the row of Fig. 3.4 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 $\text{SIG}(1,3)$ to 0, but since $3 > 0$ we do not change $\text{SIG}(2,3)$. The final signature matrix is:

	S_1	S_2	S_3	S_4
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 $\text{SIM}(S_1, S_4) = 1.0$. If we look at Fig. 3.4, 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). □

3.3.6 Exercises for Section 3.3

Exercise 3.3.1: Verify the theorem from Section 3.3.3, which relates the Jaccard similarity to the probability of minhashing to equal values, for the particular case of Fig. 3.2.

(a) Compute the Jaccard similarity of each of the pairs of columns in Fig. 3.2.

! (b) Compute, for each pair of columns of that figure, the fraction of the 120 permutations of the rows that make the two columns hash to the same value.

Exercise 3.3.2: Using the data from Fig. 3.4, add to the signatures of the columns the values of the following hash functions:

(a) $h_3(x) = 2x + 4 \mod 5$.

(b) $h_4(x) = 3x - 1 \mod 5$.

Element	S_1	S_2	S_3	S_4
0	0	1	0	1
1	0	1	0	0
2	1	0	0	1
3	0	0	1	0
4	0	0	1	1
5	1	0	0	0

Figure 3.5: Matrix for Exercise 3.3.3

Exercise 3.3.3: In Fig. 3.5 is a matrix with six rows.

(a) Compute the minhash signature for each column if we use the following three hash functions: $h_1(x) = 2x + 1 \mod 6$; $h_2(x) = 3x + 2 \mod 6$; $h_3(x) = 5x + 2 \mod 6$.

(b) Which of these hash functions are true permutations?

(c) How close are the estimated Jaccard similarities for the six pairs of columns to the true Jaccard similarities?

! **Exercise 3.3.4:** Now that we know Jaccard similarity is related to the probability that two sets minhash to the same value, reconsider Exercise 3.1.3. Can you use this relationship to simplify the problem of computing the expected Jaccard similarity of randomly chosen sets?

! **Exercise 3.3.5:** Prove that if the Jaccard similarity of two columns is 0, then minhashing always gives a correct estimate of the Jaccard similarity.

!! **Exercise 3.3.6:** One might expect that we could estimate the Jaccard similarity of columns without using all possible permutations of rows. For example, we could only allow cyclic permutations; i.e., start at a randomly chosen row r, which becomes the first in the order, followed by rows $r+1$, $r+2$, and so on, down to the last row, and then continuing with the first row, second row, and so on, down to row $r-1$. There are only n such permutations if there are n rows. However, these permutations are not sufficient to estimate the Jaccard similarity correctly. Give an example of a two-column matrix where averaging over all the cyclic permutations does not give the Jaccard similarity.

! **Exercise 3.3.7:** Suppose we want to use a MapReduce framework to compute minhash signatures. If the matrix is stored in chunks that correspond to some columns, then it is quite easy to exploit parallelism. Each Map task gets some of the columns and all the hash functions, and computes the minhash signatures of its given columns. However, suppose the matrix were chunked by rows, so that a Map task is given the hash functions and a set of rows to work on. Design Map and Reduce functions to exploit MapReduce with data in this form.

3.4 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 3.9: 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 $\binom{1,000,000}{2}$ 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 3.6 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.

Example 3.10: Figure 3.6 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.

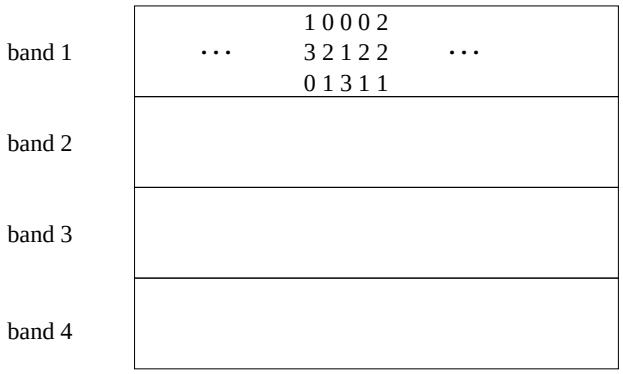

Figure 3.6: Dividing a signature matrix into four bands of three rows per band

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. □

3.4.2 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 3.3.3 that the probability 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 disagree in at least one row of a particular band is $1 - s^r$.

3. The probability that the signatures disagree in at least one row of each 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. 3.7. 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.

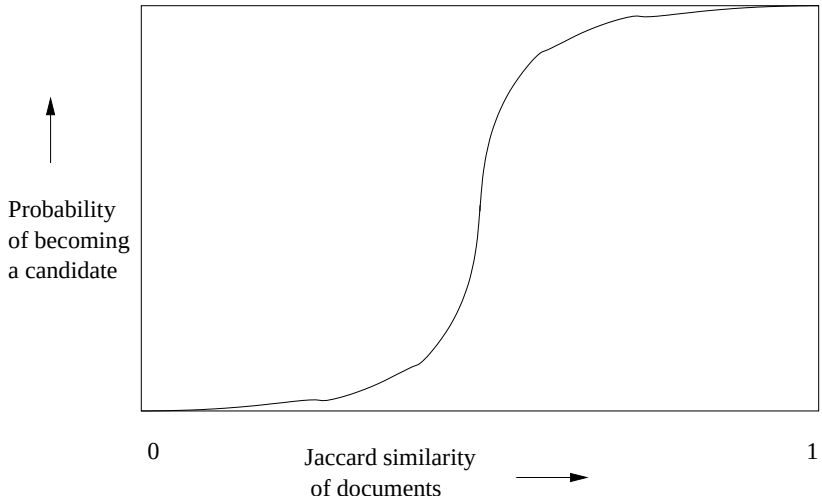

Figure 3.7: The S-curve

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 $1/2$.

Example 3.11 : 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 3.8 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 3.8: 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. □

3.4.3 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 3.3.5 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 3.4.1.
6. Examine each candidate pair's signatures and determine whether the fraction 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.

3.4.4 Exercises for Section 3.4

Exercise 3.4.1: Evaluate the S-curve $1 - (1 - s^r)^b$ for $s = 0.1, 0.2, \ldots, 0.9$, for the following values of r and b:

- $r = 3$ and $b = 10$.

- $r = 6$ and $b = 20$.
- $r = 5$ and $b = 50$.

! **Exercise 3.4.2:** For each of the (r, b) pairs in Exercise 3.4.1, compute the threshold, that is, the value of s for which the value of $1-(1-s^r)^b$ is exactly $1/2$. How does this value compare with the estimate of $(1/b)^{1/r}$ that was suggested in Section 3.4.2?

! **Exercise 3.4.3:** Use the techniques explained in Section 1.3.5 to approximate the S-curve $1 - (1 - s^r)^b$ when s^r is very small.

! **Exercise 3.4.4:** Suppose we wish to implement LSH by MapReduce. Specifically, assume chunks of the signature matrix consist of columns, and elements are key-value pairs where the key is the column number and the value is the signature itself (i.e., a vector of values).

(a) Show how to produce the buckets for all the bands as output of a single MapReduce process. *Hint*: Remember that a Map function can produce several key-value pairs from a single element.

(b) Show how another MapReduce process can convert the output of (a) to a list of pairs that need to be compared. Specifically, for each column i, there should be a list of those columns $j > i$ with which i needs to be compared.

3.5 Distance Measures

We now take a short detour to study the general notion of distance measures. The Jaccard similarity is a measure of how close sets are, although it is not really a distance measure. That is, the closer sets are, the higher the Jaccard similarity. Rather, 1 minus the Jaccard similarity is a distance measure, as we shall see; it is called the *Jaccard distance*.

However, Jaccard distance is not the only measure of closeness that makes sense. We shall examine in this section some other distance measures that have applications. Then, in Section 3.6 we see how some of these distance measures also have an LSH technique that allows us to focus on nearby points without comparing all points. Other applications of distance measures will appear when we study clustering in Chapter 7.

3.5.1 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.

3.5.2 Euclidean Distances

The most familiar distance measure is the one we normally think of as "distance." An *n-dimensional Euclidean space* is one where points are vectors of n real numbers. The conventional distance measure in this space, which we shall refer to as the *L_2-norm*, is defined:

$$d([x_1, x_2, \ldots, x_n],\ [y_1, y_2, \ldots, y_n]) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

That is, we square the distance in each dimension, sum the squares, and take the positive square root.

It is easy to verify the first three requirements for a distance measure are satisfied. The Euclidean distance between two points cannot be negative, because the positive square root is intended. Since all squares of real numbers are nonnegative, any i such that $x_i \neq y_i$ forces the distance to be strictly positive. On the other hand, if $x_i = y_i$ for all i, then the distance is clearly 0. Symmetry follows because $(x_i - y_i)^2 = (y_i - x_i)^2$. The triangle inequality requires a good deal of algebra to verify. However, it is well understood to be a property of Euclidean space: the sum of the lengths of any two sides of a triangle is no less than the length of the third side.

There are other distance measures that have been used for Euclidean spaces. For any constant r, we can define the *L_r-norm* to be the distance measure d defined by:

$$d([x_1, x_2, \ldots, x_n],\ [y_1, y_2, \ldots, y_n]) = (\sum_{i=1}^{n} |x_i - y_i|^r)^{1/r}$$

The case $r = 2$ is the usual L_2-norm just mentioned. Another common distance measure is the L_1-norm, or *Manhattan distance*. There, the distance between two points is the sum of the magnitudes of the differences in each dimension. It is called "Manhattan distance" because it is the distance one would have to

travel between points if one were constrained to travel along grid lines, as on the streets of a city such as Manhattan.

Another interesting distance measure is the L_∞-norm, which is the limit as r approaches infinity of the L_r-norm. As r gets larger, only the dimension with the largest difference matters, so formally, the L_∞-norm is defined as the maximum of $|x_i - y_i|$ over all dimensions i.

Example 3.12: Consider the two-dimensional Euclidean space (the customary plane) and the points $(2, 7)$ and $(6, 4)$. The L_2-norm gives a distance of $\sqrt{(2-6)^2 + (7-4)^2} = \sqrt{4^2 + 3^2} = 5$. The L_1-norm gives a distance of $|2-6| + |7-4| = 4 + 3 = 7$. The L_∞-norm gives a distance of

$$\max(|2-6|, |7-4|) = \max(4, 3) = 4$$

□

3.5.3 Jaccard Distance

As mentioned at the beginning of the section, we define the *Jaccard distance* of sets by $d(x, y) = 1 - \text{SIM}(x, y)$. 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 $x \cup x = x \cap x = x$. However, if $x \neq y$, then the size of $x \cap y$ is strictly less than the size of $x \cup y$, so $d(x, y)$ is strictly positive.

3. $d(x, y) = d(y, x)$ because both union and intersection are symmetric; i.e., $x \cup y = y \cup x$ and $x \cap y = y \cap x$.

4. For the triangle inequality, recall from Section 3.3.3 that $\text{SIM}(x, y)$ 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 minhash function *does not* send x and y to the same value. We can therefore translate the condition $d(x, y) \leq d(x, z) + d(z, y)$ to the statement that if h is a random minhash function, then the probability that $h(x) \neq h(y)$ is no greater than the sum of the probability that $h(x) \neq h(z)$ and the probability that $h(z) \neq h(y)$. However, this statement is true because whenever $h(x) \neq h(y)$, 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.

.

3.5.4 Cosine Distance

The *cosine distance* makes sense in spaces that have dimensions, including Euclidean spaces and discrete versions of Euclidean spaces, such as spaces where points are vectors with integer components or boolean (0 or 1) components. In such a space, points may be thought of as directions. We do not distinguish between a vector and a multiple of that vector. Then the cosine distance between two points is the angle that the vectors to those points make. This angle will be in the range 0 to 180 degrees, regardless of how many dimensions the space has.

We can calculate the cosine distance by first computing the cosine of the angle, and then applying the arc-cosine function to translate to an angle in the 0-180 degree range. Given two vectors x and y, the cosine of the angle between them is the dot product $x.y$ divided by the L_2-norms of x and y (i.e., their Euclidean distances from the origin). Recall that the dot product of vectors $[x_1, x_2, \ldots, x_n].[y_1, y_2, \ldots, y_n]$ is $\sum_{i=1}^{n} x_i y_i$.

Example 3.13: Let our two vectors be $x = [1, 2, -1]$ and $= [2, 1, 1]$. The dot product $x.y$ is $1 \times 2 + 2 \times 1 + (-1) \times 1 = 3$. The L_2-norm of both vectors is $\sqrt{6}$. For example, x has L_2-norm $\sqrt{1^2 + 2^2 + (-1)^2} = \sqrt{6}$. Thus, the cosine of the angle between x and y is $3/(\sqrt{6}\sqrt{6})$ or $1/2$. The angle whose cosine is $1/2$ is 60 degrees, so that is the cosine distance between x and y. □

We must show that the cosine distance is indeed a distance measure. We have defined it so the values are in the range 0 to 180, so no negative distances are possible. Two vectors have angle 0 if and only if they are the same direction.[4] Symmetry is obvious: the angle between x and y is the same as the angle between y and x. The triangle inequality is best argued by physical reasoning. One way to rotate from x to y is to rotate to z and thence to y. The sum of those two rotations cannot be less than the rotation directly from x to y.

3.5.5 Edit Distance

This distance makes sense when points are strings. The distance between two strings $x = x_1 x_2 \cdots x_n$ and $y = y_1 y_2 \cdots y_m$ is the smallest number of insertions and deletions of single characters that will convert x to y.

Example 3.14: 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`.

[4]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.

3. Insert `g` after `e`.

No sequence of fewer than three insertions and/or deletions will convert x to y. Thus, $d(x, y) = 3$. □

Another way to define and calculate the edit distance $d(x, y)$ is to compute a *longest common subsequence* (LCS) of x and y. An LCS of x and y is a string that is constructed by deleting positions from x and y, and that is as long as any string that can be constructed that way. The edit distance $d(x, y)$ can be calculated as the length of x plus the length of y minus twice the length of their LCS.

Example 3.15: The strings $x =$ `abcde` and $y =$ `acfdeg` from Example 3.14 have a unique LCS, which is `acde`. We can be sure it is the longest possible, because it contains every symbol appearing in both x and y. Fortunately, these common symbols appear in the same order in both strings, so we are able to use them all in an LCS. Note that the length of x is 5, the length of y is 6, and the length of their LCS is 4. The edit distance is thus $5 + 6 - 2 \times 4 = 3$, which agrees with the direct calculation in Example 3.14.

For another example, consider $x =$ `aba` and $y =$ `bab`. Their edit distance is 2. For example, we can convert x to y by deleting the first `a` and then inserting `b` at the end. There are two LCS's: `ab` and `ba`. Each can be obtained by deleting one symbol from each string. As must be the case for multiple LCS's of the same pair of strings, both LCS's have the same length. Therefore, we may compute the edit distance as $3 + 3 - 2 \times 2 = 2$. □

Edit distance is a distance measure. Surely no edit distance can be negative, and only two identical strings have an edit distance of 0. To see that edit distance is symmetric, note that a sequence of insertions and deletions can be reversed, with each insertion becoming a deletion, and vice versa. The triangle inequality is also straightforward. One way to turn a string s into a string t is to turn s into some string u and then turn u into t. Thus, the number of edits made going from s to u, plus the number of edits made going from u to t cannot be less than the smallest number of edits that will turn s into t.

3.5.6 Hamming Distance

Given a space of vectors, we define the *Hamming distance* between two vectors to be the number of components in which they differ. It should be obvious that Hamming distance is a distance measure. Clearly the Hamming distance cannot be negative, and if it is zero, then the vectors are identical. The distance does not depend on which of two vectors we consider first. The triangle inequality should also be evident. If x and z differ in m components, and z and y differ in n components, then x and y cannot differ in more than $m+n$ components. Most commonly, Hamming distance is used when the vectors are boolean; they consist of 0's and 1's only. However, in principle, the vectors can have components from any set.

Non-Euclidean Spaces

Notice that several of the distance measures introduced in this section are not Euclidean spaces. A property of Euclidean spaces that we shall find important when we take up clustering in Chapter 7 is that the average of points in a Euclidean space always exists and is a point in the space. However, consider the space of sets for which we defined the Jaccard distance. The notion of the "average" of two sets makes no sense. Likewise, the space of strings, where we can use the edit distance, does not let us take the "average" of strings.

Vector spaces, for which we suggested the cosine distance, may or may not be Euclidean. If the components of the vectors can be any real numbers, then the space is Euclidean. However, if we restrict components to be integers, then the space is not Euclidean. Notice that, for instance, we cannot find an average of the vectors $[1, 2]$ and $[3, 1]$ in the space of vectors with two integer components, although if we treated them as members of the two-dimensional Euclidean space, then we could say that their average was $[2.0, 1.5]$.

Example 3.16: The Hamming distance between the vectors 10101 and 11110 is 3. That is, these vectors differ in the second, fourth, and fifth components, while they agree in the first and third components. □

3.5.7 Exercises for Section 3.5

! **Exercise 3.5.1:** On the space of nonnegative integers, which of the following functions are distance measures? If so, prove it; if not, prove that it fails to satisfy one or more of the axioms.

(a) $\max(x, y) =$ the larger of x and y.

(b) $\text{diff}(x, y) = |x - y|$ (the absolute magnitude of the difference between x and y).

(c) $\text{sum}(x, y) = x + y$.

Exercise 3.5.2: Find the L_1 and L_2 distances between the points $(5, 6, 7)$ and $(8, 2, 4)$.

!! **Exercise 3.5.3:** Prove that if i and j are any positive integers, and $i < j$, then the L_i norm between any two points is greater than the L_j norm between those same two points.

Exercise 3.5.4: Find the Jaccard distances between the following pairs of sets:

(a) $\{1, 2, 3, 4\}$ and $\{2, 3, 4, 5\}$.

(b) $\{1, 2, 3\}$ and $\{4, 5, 6\}$.

Exercise 3.5.5: Compute the cosines of the angles between each of the following pairs of vectors.[5]

(a) $(3, -1, 2)$ and $(-2, 3, 1)$.

(b) $(1, 2, 3)$ and $(2, 4, 6)$.

(c) $(5, 0, -4)$ and $(-1, -6, 2)$.

(d) $(0, 1, 1, 0, 1, 1)$ and $(0, 0, 1, 0, 0, 0)$.

! **Exercise 3.5.6:** Prove that the cosine distance between any two vectors of 0's and 1's, of the same length, is at most 90 degrees.

Exercise 3.5.7: Find the edit distances (using only insertions and deletions) between the following pairs of strings.

(a) `abcdef` and `bdaefc`.

(b) `abccdabc` and `acbdcab`.

(c) `abcdef` and `baedfc`.

! **Exercise 3.5.8:** There are a number of other notions of edit distance available. For instance, we can allow, in addition to insertions and deletions, the following operations:

i. *Mutation*, where one symbol is replaced by another symbol. Note that a mutation can always be performed by an insertion followed by a deletion, but if we allow mutations, then this change counts for only 1, not 2, when computing the edit distance.

ii. *Transposition*, where two adjacent symbols have their positions swapped. Like a mutation, we can simulate a transposition by one insertion followed by one deletion, but here we count only 1 for these two steps.

Repeat Exercise 3.5.7 if edit distance is defined to be the number of insertions, deletions, mutations, and transpositions needed to transform one string into another.

! **Exercise 3.5.9:** Prove that the edit distance discussed in Exercise 3.5.8 is indeed a distance measure.

Exercise 3.5.10: Find the Hamming distances between each pair of the following vectors: 000000, 110011, 010101, and 011100.

[5]Note that what we are asking for is not precisely the cosine distance, but from the cosine of an angle, you can compute the angle itself, perhaps with the aid of a table or library function.

Exercise 3.7.5: Suppose we have points in a 3-dimensional Euclidean space: $p_1 = (1, 2, 3)$, $p_2 = (0, 2, 4)$, and $p_3 = (4, 3, 2)$. Consider the three hash functions defined by the three axes (to make our calculations very easy). Let buckets be of length a, with one bucket the interval $[0, a)$ (i.e., the set of points x such that $0 \leq x < a$), the next $[a, 2a)$, the previous one $[-a, 0)$, and so on.

(a) For each of the three lines, assign each of the points to buckets, assuming $a = 1$.

(b) Repeat part (a), assuming $a = 2$.

(c) What are the candidate pairs for the cases $a = 1$ and $a = 2$?

! (d) For each pair of points, for what values of a will that pair be a candidate pair?

3.8 Applications of Locality-Sensitive Hashing

In this section, we shall explore three examples of how LSH is used in practice. In each case, the techniques we have learned must be modified to meet certain constraints of the problem. The three subjects we cover are:

1. *Entity Resolution*: This term refers to matching data records that refer to the same real-world entity, e.g., the same person. The principal problem addressed here is that the similarity of records does not match exactly either the similar-sets or similar-vectors models of similarity on which the theory is built.

2. *Matching Fingerprints*: It is possible to represent fingerprints as sets. However, we shall explore a different family of locality-sensitive hash functions from the one we get by minhashing.

3. *Matching Newspaper Articles*: Here, we consider a different notion of shingling that focuses attention on the core article in an on-line newspaper's Web page, ignoring all the extraneous material such as ads and newspaper-specific material.

3.8.1 Entity Resolution

It is common to have several data sets available, and to know that they refer to some of the same entities. For example, several different bibliographic sources provide information about many of the same books or papers. In the general case, we have records describing entities of some type, such as people or books. The records may all have the same format, or they may have different formats, with different kinds of information.

There are many reasons why information about an entity may vary, even if the field in question is supposed to be the same. For example, names may be

expressed differently in different records because of misspellings, absence of a middle initial, use of a nickname, and many other reasons. For example, "Bob S. Jomes" and "Robert Jones Jr." may or may not be the same person. If records come from different sources, the fields may differ as well. One source's records may have an "age" field, while another does not. The second source might have a "date of birth" field, or it may have no information at all about when a person was born.

3.8.2 An Entity-Resolution Example

We shall examine a real example of how LSH was used to deal with an entity-resolution problem. Company A was engaged by Company B to solicit customers for B. Company B would pay A a yearly fee, as long as the customer maintained their subscription. They later quarreled and disagreed over how many customers A had provided to B. Each had about 1,000,000 records, some of which described the same people; those were the customers A had provided to B. The records had different data fields, but unfortunately none of those fields was "this is a customer that A had provided to B." Thus, the problem was to match records from the two sets to see if a pair represented the same person.

Each record had fields for the name, address, and phone number of the person. However, the values in these fields could differ for many reasons. Not only were there the misspellings and other naming differences mentioned in Section 3.8.1, but there were other opportunities to disagree as well. A customer might give their home phone to A and their cell phone to B. Or they might move, and tell B but not A (because they no longer had need for a relationship with A). Area codes of phones sometimes change.

The strategy for identifying records involved scoring the differences in three fields: name, address, and phone. To create a *score* describing the likelihood that two records, one from A and the other from B, described the same person, 100 points was assigned to each of the three fields, so records with exact matches in all three fields got a score of 300. However, there were deductions for mismatches in each of the three fields. As a first approximation, edit-distance (Section 3.5.5) was used, but the penalty grew quadratically with the distance. Then, certain publicly available tables were used to reduce the penalty in appropriate situations. For example, "Bill" and "William" were treated as if they differed in only one letter, even though their edit-distance is 5.

However, it is not feasible to score all one trillion pairs of records. Thus, a simple LSH was used to focus on likely candidates. Three "hash functions" were used. The first sent records to the same bucket only if they had identical names; the second did the same but for identical addresses, and the third did the same for phone numbers. In practice, there was no hashing; rather the records were sorted by name, so records with identical names would appear consecutively and get scored for overall similarity of the name, address, and phone. Then the records were sorted by address, and those with the same

When Are Record Matches Good Enough?

While every case will be different, it may be of interest to know how the experiment of Section 3.8.3 turned out on the data of Section 3.8.2. For scores down to 185, the value of x was very close to 10; i.e., these scores indicated that the likelihood of the records representing the same person was essentially 1. Note that a score of 185 in this example represents a situation where one field is the same (as would have to be the case, or the records would never even be scored), one field was completely different, and the third field had a small discrepancy. Moreover, for scores as low as 115, the value of x was noticeably less than 45, meaning that some of these pairs did represent the same person. Note that a score of 115 represents a case where one field is the same, but there is only a slight similarity in the other two fields.

address were scored. Finally, the records were sorted a third time by phone, and records with identical phones were scored.

This approach missed a record pair that truly represented the same person but none of the three fields matched exactly. Since the goal was to prove in a court of law that the persons were the same, it is unlikely that such a pair would have been accepted by a judge as sufficiently similar anyway.

3.8.3 Validating Record Matches

What remains is to determine how high a score indicates that two records truly represent the same individual. In the example at hand, there was an easy way to make that decision, and the technique can be applied in many similar situations. It was decided to look at the creation-dates for the records at hand, and to assume that 90 days was an absolute maximum delay between the time the service was bought at Company A and registered at B. Thus, a proposed match between two records that were chosen at random, subject only to the constraint that the date on the B-record was between 0 and 90 days after the date on the A-record, would have an average delay of 45 days.

It was found that of the pairs with a perfect 300 score, the average delay was 10 days. If you assume that 300-score pairs are surely correct matches, then you can look at the pool of pairs with any given score s, and compute the average delay of those pairs. Suppose that the average delay is x, and the fraction of true matches among those pairs with score s is f. Then $x = 10f + 45(1 - f)$, or $x = 45 - 35f$. Solving for f, we find that the fraction of the pairs with score s that are truly matches is $(45 - x)/35$.

The same trick can be used whenever:

1. There is a scoring system used to evaluate the likelihood that two records

represent the same entity, and

2. There is some field, not used in the scoring, from which we can derive a measure that differs, on average, for true pairs and false pairs.

For instance, suppose there were a "height" field recorded by both companies A and B in our running example. We can compute the average difference in height for pairs of random records, and we can compute the average difference in height for records that have a perfect score (and thus surely represent the same entities). For a given score s, we can evaluate the average height difference of the pairs with that score and estimate the probability of the records representing the same entity. That is, if h_0 is the average height difference for the perfect matches, h_1 is the average height difference for random pairs, and h is the average height difference for pairs of score s, then the fraction of good pairs with score s is $(h_1 - h)/(h_1 - h_0)$.

3.8.4 Matching Fingerprints

When fingerprints are matched by computer, the usual representation is not an image, but a set of locations in which *minutiae* are located. A minutia, in the context of fingerprint descriptions, is a place where something unusual happens, such as two ridges merging or a ridge ending. If we place a grid over a fingerprint, we can represent the fingerprint by the set of grid squares in which minutiae are located.

Ideally, before overlaying the grid, fingerprints are normalized for size and orientation, so that if we took two images of the same finger, we would find minutiae lying in exactly the same grid squares. We shall not consider here the best ways to normalize images. Let us assume that some combination of techniques, including choice of grid size and placing a minutia in several adjacent grid squares if it lies close to the border of the squares enables us to assume that grid squares from two images have a significantly higher probability of agreeing in the presence or absence of a minutia than if they were from images of different fingers.

Thus, fingerprints can be represented by sets of grid squares – those where their minutiae are located – and compared like any sets, using the Jaccard similarity or distance. There are two versions of fingerprint comparison, however.

- The *many-one* problem is the one we typically expect. A fingerprint has been found on a gun, and we want to compare it with all the fingerprints in a large database, to see which one matches.
- The *many-many* version of the problem is to take the entire database, and see if there are any pairs that represent the same individual.

While the many-many version matches the model that we have been following for finding similar items, the same technology can be used to speed up the many-one problem.

3.8.5 A LSH Family for Fingerprint Matching

We could minhash the sets that represent a fingerprint, and use the standard LSH technique from Section 3.4. However, since the sets are chosen from a relatively small set of grid points (perhaps 1000), the need to minhash them into more succinct signatures is not clear. We shall study here another form of locality-sensitive hashing that works well for data of the type we are discussing.

Suppose for an example that the probability of finding a minutia in a random grid square of a random fingerprint is 20%. Also, assume that if two fingerprints come from the same finger, and one has a minutia in a given grid square, then the probability that the other does too is 80%. We can define a locality-sensitive family of hash functions as follows. Each function f in this family $\mathbf{F}$ is defined by three grid squares. Function f says "yes" for two fingerprints if both have minutiae in all three grid squares, and otherwise f says "no." Put another way, we may imagine that f sends to a single bucket all fingerprints that have minutiae in all three of f's grid points, and sends each other fingerprint to a bucket of its own. In what follows, we shall refer to the first of these buckets as "the" bucket for f and ignore the buckets that are required to be singletons.

If we want to solve the many-one problem, we can use many functions from the family $\mathbf{F}$ and precompute their buckets of fingerprints to which they answer "yes." Then, given a new fingerprint that we want to match, we determine which of these buckets it belongs to and compare it with all the fingerprints found in any of those buckets. To solve the many-many problem, we compute the buckets for each of the functions and compare all fingerprints in each of the buckets.

Let us consider how many functions we need to get a reasonable probability of catching a match, without having to compare the fingerprint on the gun with each of the millions of fingerprints in the database. First, the probability that two fingerprints from different fingers would be in the bucket for a function f in $\mathbf{F}$ is $(0.2)^6 = 0.000064$. The reason is that they will both go into the bucket only if they each have a minutia in each of the three grid points associated with f, and the probability of each of those six independent events is 0.2.

Now, consider the probability that two fingerprints from the same finger wind up in the bucket for f. The probability that the first fingerprint has minutiae in each of the three squares belonging to f is $(0.2)^3 = 0.008$. However, if it does, then the probability is $(0.8)^3 = 0.512$ that the other fingerprint will as well. Thus, if the fingerprints are from the same finger, there is a $0.008 \times 0.512 = 0.004096$ probability that they will both be in the bucket of f. That is not much; it is about one in 200. However, if we use many functions from $\mathbf{F}$, but not too many, then we can get a good probability of matching fingerprints from the same finger while not having too many false positives – fingerprints that must be considered but do not match.

Example 3.22: For a specific example, let us suppose that we use 1024 functions chosen randomly from $\mathbf{F}$. Next, we shall construct a new family $\mathbf{F}_1$ by performing a 1024-way OR on $\mathbf{F}$. Then the probability that $\mathbf{F}_1$

will put fingerprints from the same finger together in at least one bucket is $1 - (1 - 0.004096)^{1024} = 0.985$. On the other hand, the probability that two fingerprints from different fingers will be placed in the same bucket is $(1 - (1 - 0.000064)^{1024} = 0.063$. That is, we get about 1.5% false negatives and about 6.3% false positives. □

The result of Example 3.22 is not the best we can do. While it offers only a 1.5% chance that we shall fail to identify the fingerprint on the gun, it does force us to look at 6.3% of the entire database. Increasing the number of functions from **F** will increase the number of false positives, with only a small benefit of reducing the number of false negatives below 1.5%. On the other hand, we can also use the AND construction, and in so doing, we can greatly reduce the probability of a false positive, while making only a small increase in the false-negative rate. For instance, we could take 2048 functions from **F** in two groups of 1024. Construct the buckets for each of the functions. However, given a fingerprint P on the gun:

1. Find the buckets from the first group in which P belongs, and take the union of these buckets.
2. Do the same for the second group.
3. Take the intersection of the two unions.
4. Compare P only with those fingerprints in the intersection.

Note that we still have to take unions and intersections of large sets of fingerprints, but we compare only a small fraction of those. It is the comparison of fingerprints that takes the bulk of the time; in steps (1) and (2) fingerprints can be represented by their integer indices in the database.

If we use this scheme, the probability of detecting a matching fingerprint is $(0.985)^2 = 0.970$; that is, we get about 3% false negatives. However, the probability of a false positive is $(0.063)^2 = 0.00397$. That is, we only have to examine about 1/250th of the database.

3.8.6 Similar News Articles

Our last case study concerns the problem of organizing a large repository of on-line news articles by grouping together Web pages that were derived from the same basic text. It is common for organizations like The Associated Press to produce a news item and distribute it to many newspapers. Each newspaper puts the story in its on-line edition, but surrounds it by information that is special to that newspaper, such as the name and address of the newspaper, links to related articles, and links to ads. In addition, it is common for the newspaper to modify the article, perhaps by leaving off the last few paragraphs or even deleting text from the middle. As a result, the same news article can appear quite different at the Web sites of different newspapers.

The problem looks very much like the one that was suggested in Section 3.4: find documents whose shingles have a high Jaccard similarity. Note that this problem is different from the problem of finding news articles that tell about the same events. The latter problem requires other techniques, typically examining the set of important words in the documents (a concept we discussed briefly in Section 1.3.1) and clustering them to group together different articles about the same topic.

However, an interesting variation on the theme of shingling was found to be more effective for data of the type described. The problem is that shingling as we described it in Section 3.2 treats all parts of a document equally. However, we wish to ignore parts of the document, such as ads or the headlines of other articles to which the newspaper added a link, that are not part of the news article. It turns out that there is a noticeable difference between text that appears in prose and text that appears in ads or headlines. Prose has a much greater frequency of stop words, the very frequent words such as "the" or "and." The total number of words that are considered stop words varies with the application, but it is common to use a list of several hundred of the most frequent words.

Example 3.23: A typical ad might say simply "Buy Sudzo." On the other hand, a prose version of the same thought that might appear in an article is "I recommend that you buy Sudzo for your laundry." In the latter sentence, it would be normal to treat "I," "that," "you," "for," and "your" as stop words. □

Suppose we define a *shingle* to be a stop word followed by the next two words. Then the ad "Buy Sudzo" from Example 3.23 has no shingles and would not be reflected in the representation of the Web page containing that ad. On the other hand, the sentence from Example 3.23 would be represented by five shingles: "I recommend that," "that you buy," "you buy Sudzo," "for your laundry," and "your laundry x," where x is whatever word follows that sentence.

Suppose we have two Web pages, each of which consists of half news text and half ads or other material that has a low density of stop words. If the news text is the same but the surrounding material is different, then we would expect that a large fraction of the shingles of the two pages would be the same. They might have a Jaccard similarity of 75%. However, if the surrounding material is the same but the news content is different, then the number of common shingles would be small, perhaps 25%. If we were to use the conventional shingling, where shingles are (say) sequences of 10 consecutive characters, we would expect the two documents to share half their shingles (i.e., a Jaccard similarity of 1/3), regardless of whether it was the news or the surrounding material that they shared.

3.8.7 Exercises for Section 3.8

Exercise 3.8.1: Suppose we are trying to perform entity resolution among bibliographic references, and we score pairs of references based on the similarities of their titles, list of authors, and place of publication. Suppose also that all references include a year of publication, and this year is equally likely to be any of the ten most recent years. Further, suppose that we discover that among the pairs of references with a perfect score, there is an average difference in the publication year of 0.1.[6] Suppose that the pairs of references with a certain score s are found to have an average difference in their publication dates of 2. What is the fraction of pairs with score s that truly represent the same publication? *Note*: Do not make the mistake of assuming the average difference in publication date between random pairs is 5 or 5.5. You need to calculate it exactly, and you have enough information to do so.

Exercise 3.8.2: Suppose we use the family $\mathbf{F}$ of functions described in Section 3.8.5, where there is a 20% chance of a minutia in an grid square, an 80% chance of a second copy of a fingerprint having a minutia in a grid square where the first copy does, and each function in $\mathbf{F}$ being formed from three grid squares. In Example 3.22, we constructed family $\mathbf{F}_1$ by using the OR construction on 1024 members of $\mathbf{F}$. Suppose we instead used family $\mathbf{F}_2$ that is a 2048-way OR of members of $\mathbf{F}$.

(a) Compute the rates of false positives and false negatives for $\mathbf{F}_2$.

(b) How do these rates compare with what we get if we organize the same 2048 functions into a 2-way AND of members of $\mathbf{F}_1$, as was discussed at the end of Section 3.8.5?

Exercise 3.8.3: Suppose fingerprints have the same statistics outlined in Exercise 3.8.2, but we use a base family of functions $\mathbf{F}'$ defined like $\mathbf{F}$, but using only two randomly chosen grid squares. Construct another set of functions $\mathbf{F}'_1$ from $\mathbf{F}'$ by taking the n-way OR of functions from $\mathbf{F}'$. What, as a function of n, are the false positive and false negative rates for $\mathbf{F}'_1$?

Exercise 3.8.4: Suppose we use the functions $\mathbf{F}_1$ from Example 3.22, but we want to solve the many-many problem.

(a) If two fingerprints are from the same finger, what is the probability that they will not be compared (i.e., what is the false negative rate)?

(b) What fraction of the fingerprints from different fingers will be compared (i.e., what is the false positive rate)?

! **Exercise 3.8.5:** Assume we have the set of functions $\mathbf{F}$ as in Exercise 3.8.2, and we construct a new set of functions $\mathbf{F}_3$ by an n-way OR of functions in $\mathbf{F}$. For what value of n is the sum of the false positive and false negative rates minimized?

[6] We might expect the average to be 0, but in practice, errors in publication year do occur.