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.

The naive approach to finding pairs of similar items requires us to look at every pair of items. When we are dealing with a large dataset, looking at all pairs of items may be prohibitive, even given an abundance of hardware resources. For example, even a million items gives us half a trillion pairs to examine, and a million items is considered a "small" dataset by today's standards.

It is therefore a pleasant surprise to learn of a family of techniques called *locality-sensitive hashing*, or *LSH*, that allows us to focus on pairs that are likely to be similar, without having to look at all pairs. Thus, it is possible that we can avoid the quadratic growth in computation time that is required by the naive algorithm. There is usually a downside to locality-sensitive hashing, due to the presence of false negatives, that is, pairs of items that are similar, yet are not included in the set of pairs that we examine, but by careful tuning we can reduce the fraction of false negatives by increasing the number of pairs we consider.

The general idea behind LSH is that we hash items using many different hash functions. These hash functions are not the conventional sort of hash functions. Rather, they are carefully designed to have the property that pairs are much more likely to wind up in the same bucket of a hash function if the items are similar than if they are not similar. We then can examine only the *candidate pairs*, which are pairs of items that wind up in the same bucket for at least one of the hash functions.

We begin our discussion of LSH with an examination of the problem of finding similar documents – those that share a lot of common text. We first show how to convert documents into sets (Section 3.2) in a way that lets us view textual similarity of documents as sets having a large overlap. More precisely,

we measure the similarity of sets by their *Jaccard similarity*, the ratio of the sizes of their intersection and union. A second key trick we need is *minhashing* (Section 3.3), which is a way to convert large sets into much smaller representations, called *signatures*, that still enable us to estimate closely the Jaccard similarity of the represented sets. Finally, in Section 3.4 we see how to apply the bucketing idea inherent in LSH to the signatures.

In Section 3.5 we begin our study of how to apply LSH to items other than sets. We consider the general notion of a distance measure that tells to what degree items are similar. Then, in Section 3.6 we consider the general idea of locality-sensitive hashing, and in Section 3.7 we see how to do LSH for some data types other than sets. Then, Section 3.8 examines in detail several applications of the LSH idea. Finally, we consider in Section 3.9 some techniques for finding similar sets that can be more efficient than LSH when the degree of similarity we want is very high.

3.1 Applications of Set Similarity

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, which is 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 the technique called shingling, which is the subject of 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, $\text{SIM}(S, T) = 3/8$. □

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

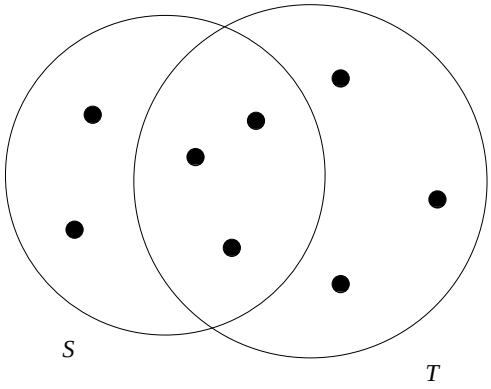

Figure 3.1: Two sets with Jaccard similarity 3/8

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 much 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 reuse of Web pages from one academic 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.

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.

[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.

Movie Ratings

Netflix records which movies each of its customers rented, and also the ratings assigned to those movies by the customers. We can regard 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 "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 use 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. □

[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 each of 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 also gives a reasonable measure of bag similarity.

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 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 $\{$`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. □

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 elements. 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 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 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

[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.

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.

The remarkable thing about signature matrices is that we can use their columns to estimate the Jaccard similarity of the sets that correspond to the columns of signature matrix. By the theorem proved in Section 3.3.3, we know that the probability that two columns have the same value in a given row of the signature matrix equals the Jaccard similarity of the sets corresponding to those columns. Moreover, since the permutations on which the minhash values are based were chosen independently, we can think of each row of the signature matrix as an independent experiment. Thus, the expected number of rows in which two columns agree equals the Jaccard similarity of their corresponding sets. Moreover, the more minhashings we use, i.e., the more rows in the signature matrix, the smaller the expected error in the estimate of the Jaccard similarity will be.

3.3.5 Computing Minhash Signatures in Practice

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 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 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 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
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 Speeding Up Minhashing

The process of minhashing is time-consuming, since we need to examine the entire k-row matrix M for each minhash function we want. Let us first return

to the model of Section 3.3.2, where we imagine rows are actually permuted. But to compute one minhash function on all the columns, we shall not go all the way to the end of the permutation, but only look at the first m out of k rows. If we make m small compared with k, we reduce the work by a large factor, k/m.

However, there is a downside to making m small. As long as each column has at least one 1 in the first m rows in permuted order, the rows after the mth have no effect on any minhash value and may as well not be looked at. But what if some columns are all-0's in the first m rows? We have no minhash value for those columns, and will instead have to use a special symbol, for which we shall use ∞.

When we examine the minhash signatures of two columns in order to estimate the Jaccard similarity of their underlying sets, as in Section 3.3.4, we have to take into account the possibility that one or both columns have ∞ as their minhash value for some components of the signature. There are three cases:

1. If neither column has ∞ in a given row, then there is no change needed. Count this row as an example of equal values if the two values are the same, and as an example of unequal values if not.

2. One column has ∞ and the other does not. In this case, had we used all the rows of the original permuted matrix M, the column that has the ∞ would eventually have been given some row number, and that number will surely not be one of the first m rows in the permuted order. But the other column *does* have a value that is one of the first m rows. Thus, we surely have an example of unequal minhash values, and we count this row of the signature matrix as such an example.

3. Now, suppose both columns have ∞ in row. Then in the original permuted matrix M, the first m rows of both columns were all 0's. We thus have no information about the Jaccard similarity of the corresponding sets; that similarity is only a function of the last $k - m$ rows, which we have chosen not to look at. We therefore count this row of the signature matrix as neither an example of equal values nor of unequal values.

As long as the third case, where both columns have ∞, is rare, we get almost as many examples to average as there are rows in the signature matrix. That effect will reduce the accuracy of our estimates of the Jaccard distance somewhat, but not much. And since we are now able to compute minhash values for all the columns much faster than if we examined all the rows of M, we can afford the time to apply a few more minhash functions. We get even better accuracy than originally, and we do so faster than before.

3.3.7 Speedup Using Hash Functions

As before, there are reasons not to physically permute rows in the manner assumed in Section 3.3.6. However, the idea of true permutations makes more

sense in the context of Section 3.3.6 than it did in Section 3.3.2. The reason is that we do not need to construct a full permutation of k elements, but only pick a small number m out of the k rows and then pick a random permutation of those rows. Depending on the value of m and how the matrix M is stored, it might make sense to follow the algorithm suggested by Section 3.3.6 literally.

However, it is more likely that a strategy akin to Section 3.3.5 is needed. Now, the rows of M are fixed, and not permuted. We choose a hash function that hashes row numbers, and compute hash values for only the first m rows. That is, we follow the algorithm of Section 3.3.5, but only until we reach the mth row, whereupon we stop and, and for each columns, we take the minimum hash value seen so far as the minhash value for that column.

Since some column may have 0 in all m rows, it is possible that some of the minhash values will be ∞. Assuming m is sufficiently large that ∞ minhash values are rare, we still get a good estimate of the Jaccard similarity of sets by comparing columns of the signature matrix. Suppose T is the set of elements of the universal set that are represented by the first m rows of matrix M. Let S_1 and S_2 be the sets represented by two columns of M. Then the first m rows of M represent the sets $S_1 \cap T$ and $S_2 \cap T$. If both these sets are empty (i.e., both columns are all-0 in their first m rows), then this minhash function will be ∞ in both columns and will be ignored when estimating the Jaccard similarity of the columns' underlying sets.

If at least one of the sets $S_1 \cap T$ and $S_2 \cap T$ is nonempty, then the probability of the two columns having equal values for this minhash function is the Jaccard similarity of these two sets, that is

$$\frac{|S_1 \cap S_2 \cap T|}{|(S_1 \cup S_2) \cap T|}$$

As long as T is chosen to be a random subset of the universal set, the expected value of this fraction will be the same as the Jaccard similarity of S_1 and S_2. However, there will be some random variation, since depending on T, we could find more or less than an average number of type X rows (1's in both columns) and/or type Y rows (1 in one column and 0 in the other) among the first m rows of matrix M.

To mitigate this variation, we do not use the same set T for each minhashing that we do. Rather, we divide the rows of M into k/m groups.[4] Then for each hash function, we compute one minhash value by examining only the first m rows of M, a different minhash value by examining only the second m rows, and so on. We thus get k/m minhash values from a single hash function and a single pass over all the rows of M. In fact, if k/m is large enough, we may get all the rows of the signature matrix that we need by a single hash function applied to each of the subsets of rows of M.

[4]In what follows, we assume m divides k evenly, for convenience. It is unimportant, as long as k/m is large, if some rows are not included in any group because k is not an integer multiple of m.

Moreover, by using each of the rows of M to compute one of these minhash values, we tend to balance out the errors in estimation of the Jaccard similarity due to any one particular subset of the rows. That is, the Jaccard similarity of S_1 and S_2 determines the ratio of type X and type Y rows. All the type X rows are distributed among the k/m sets of rows, and likewise the type Y rows. Thus, while one set of m rows may have more of one type of row than average, there must then be some other set of m rows with fewer than average of that same type.

Example 3.9 : In Fig. 3.5 we see a matrix representing three sets S_1, S_2, and S_3, with a universal set of eight elements; i.e., $k = 8$. Let us pick $m = 4$, so one pass through the rows yields two minhash values, one based on the first four rows and the other on the second four rows.

S_1	S_2	S_3
0	0	0
0	0	0
0	0	1
0	1	1
1	1	1
1	1	0
1	0	0
0	0	0

Figure 3.5: A Boolean matrix representing three sets

First, note that the Jaccard similarities of the three sets are $\text{SIM}(S_1, S_2) = 1/2$, $\text{SIM}(S_1, S_3) = 1/5$, and $\text{SIM}(S_2, S_3) = 1/2$. Now, look at the first four rows only. Whatever hash function we use, the minhash value for S_1 will be ∞, the minhash value for S_2 will be the hash value of the 4th row, and the minhash value for S_3 will be the smaller of the hash values for the third and fourth rows. Thus, the minhash values for S_1 and S_2 will never agree. That makes sense, since if T is the set of elements represented by the first four rows, then $S_1 \cap T = \emptyset$, and therefore $\text{SIM}(S_1 \cap T, S_2 \cap T) = 0$. However, in the second four rows, the Jaccard similarity of S_1 and S_2 restricted to the elements represented by the last four rows is $2/3$.

We conclude that if we generate signatures consisting of two minhash values using this hash function, one based on the first four rows and the second based on the last four rows, the expected number of matches we get between the signatures for S_1 and S_2 is the average of 0 and $2/3$, or $1/3$. Since the actual Jaccard similarity of S_1 and S_2 is $1/2$, there is an error, but not too great an error. In larger examples, where minhash values are based on far more than four rows, the expected error will approach zero.

Similarly, we can see the effect of splitting the rows on the other two pairs of columns. Between S_1 and S_3, the top half represents sets with a Jaccard

similarity of 0, while the bottom half represents sets with a Jaccard similarity 1/3. The expected number of matches in the signatures of S_1 and S_3 is therefore the average of these, or 1/6. That compares with the true Jaccard similarity $\text{SIM}(S_1, S_3) = 1/5$. Finally, when we compare S_2 and S_3, we note that the Jaccard similarity of these columns in the first four rows is 1/2, and so is their Jaccard similarity in the bottom four rows. The average, 1/2, also agrees exactly with $\text{SIM}(S_2, S_3) = 1/2$. □

3.3.8 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 3.6: Matrix for Exercise 3.3.3

Exercise 3.3.3: In Fig. 3.6 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.

! **Exercise 3.3.8:** As we noticed in Section 3.3.6, we have problems when a column has only 0's. If we compute a minhash function using entire columns (as in Section 3.3.2), then the only time we get all 0's in a column is if that column represents the empty set. How should we handle the empty set to make sure no errors in Jaccard-similarity estimation are introduced?

!! **Exercise 3.3.9:** In Example 3.9, each of the three estimates of Jaccard similarity we obtained was either smaller than or the same as the true Jaccard similarity. Is it possible that for another pair of columns, the average of the Jaccard similarities of the upper and lower halves will exceed the actual Jaccard similarity of the columns?

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 3.10: 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 3.11: Figure 3.7 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

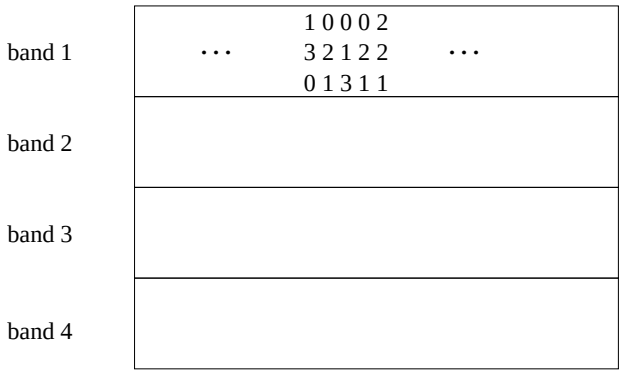

Figure 3.7: Dividing a signature matrix into four bands of three rows per band

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. □

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 $1 - (1 - s^r)^b$.

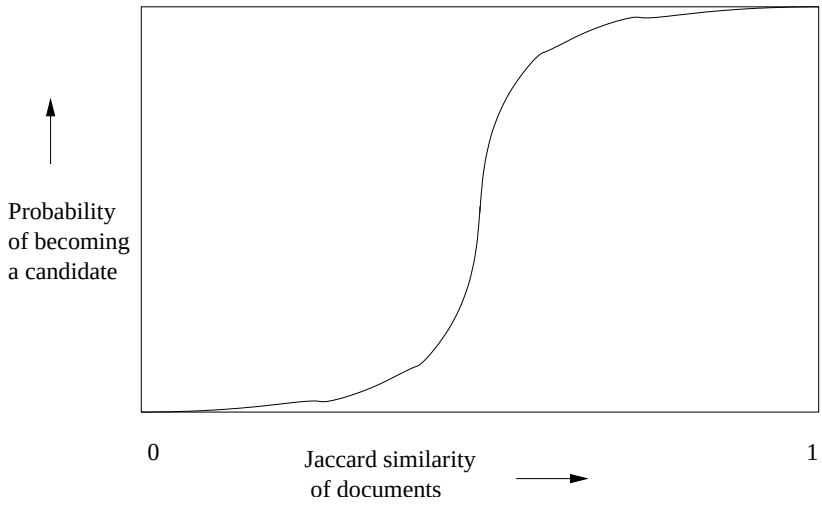

Figure 3.8: The S-curve

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. 3.8. 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 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.

Example 3.12 : 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 3.9 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.

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

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

Figure 3.9: Values of the S-curve for $b = 20$ and $r = 5$

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 3.13: 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 3.14: 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.[5] 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.

[5]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.

3.5.5 Edit Distance

This distance makes sense when points are strings. The distance between two strings $x = x_1x_2 \cdots x_n$ and $y = y_1y_2 \cdots y_m$ is the smallest number of insertions and deletions of single characters that will convert x to y.

Example 3.15: 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, $d(x, y) = 3$. □

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 3.16: The strings $x =$ `abcde` and $y =$ `acfdeg` from Example 3.15 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 3.15.

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.

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 $[2.0, 1.5]$.

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.

Example 3.17: 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.[6]

(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:

[6]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.

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.

3.6 The Theory of Locality-Sensitive Functions

The LSH technique developed in Section 3.4 is one example of a family of functions (the minhash functions) that can be combined (by the banding technique) to distinguish strongly between pairs at a low distance from pairs at a high distance. The steepness of the S-curve in Fig. 3.8 reflects how effectively we can avoid false positives and false negatives among the candidate pairs.

Now, we shall explore other families of functions, besides the minhash functions, that can serve to produce candidate pairs efficiently. These functions can apply to the space of sets and the Jaccard distance, or to another space and/or another distance measure. There are three conditions that we need for a family of functions:

1. They must be more likely to make close pairs be candidate pairs than distant pairs. We make this notion precise in Section 3.6.1.
2. They must be statistically independent, in the sense that it is possible to estimate the probability that two or more functions will all give a certain response by the product rule for independent events.
3. They must be efficient, in two ways:
   (a) They must be able to identify candidate pairs in time much less than the time it takes to look at all pairs. For example, minhash functions have this capability, since we can hash sets to minhash values in time proportional to the size of the data, rather than the square of the number of sets in the data. Since sets with common values are colocated in a bucket, we have implicitly produced the

candidate pairs for a single minhash function in time much less than the number of pairs of sets.

(b) They must be combinable to build functions that are better at avoiding false positives and negatives, and the combined functions must also take time that is much less than the number of pairs. For example, the banding technique of Section 3.4.1 takes single minhash functions, which satisfy condition 3a but do not, by themselves have the S-curve behavior we want, and produces from a number of minhash functions a combined function that has the S-curve shape.

Our first step is to define "locality-sensitive functions" generally. We then see how the idea can be applied in several applications. Finally, we discuss how to apply the theory to arbitrary data with either a cosine distance or a Euclidean distance measure.

3.6.1 Locality-Sensitive Functions

For the purposes of this section, we shall consider functions that take two items and render a decision about whether these items should be a candidate pair. In many cases, the function f will "hash" items, and the decision will be based on whether or not the result is equal. Because it is convenient to use the notation $f(x) = f(y)$ to mean that $f(x, y)$ is "yes; make x and y a candidate pair," we shall use $f(x) = f(y)$ as a shorthand with this meaning. We also use $f(x) \neq f(y)$ to mean "do not make x and y a candidate pair unless some other function concludes we should do so."

A collection of functions of this form will be called a *family* of functions. For example, the family of minhash functions, each based on one of the possible permutations of rows of a characteristic matrix, form a family.

Let $d_1 < d_2$ be two distances according to some distance measure d. A family $\mathbf{F}$ of functions is said to be (d_1, d_2, p_1, p_2)-*sensitive* if for every f in $\mathbf{F}$:

1. If $d(x, y) \leq d_1$, then the probability that $f(x) = f(y)$ is at least p_1.
2. If $d(x, y) \geq d_2$, then the probability that $f(x) = f(y)$ is at most p_2.

Figure 3.10 illustrates what we expect about the probability that a given function in a (d_1, d_2, p_1, p_2)-sensitive family will declare two items to be a candidate pair. Notice that we say nothing about what happens when the distance between the items is strictly between d_1 and d_2, but we can make d_1 and d_2 as close as we wish. The penalty is that typically p_1 and p_2 are then close as well. As we shall see, it is possible to drive p_1 and p_2 apart while keeping d_1 and d_2 fixed.

3.6.2 Locality-Sensitive Families for Jaccard Distance

For the moment, we have only one way to find a family of locality-sensitive functions: use the family of minhash functions, and assume that the distance

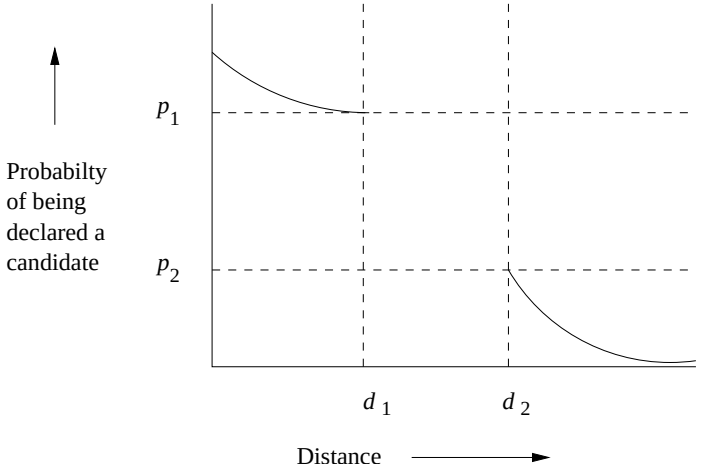

Figure 3.10: Behavior of a (d_1, d_2, p_1, p_2)-sensitive function

measure is the Jaccard distance. As before, we interpret a minhash function h to make x and y a candidate pair if and only if $h(x) = h(y)$.

- The family of minhash functions is a $(d_1, d_2, 1-d_1, 1-d_2)$-sensitive family for any d_1 and d_2, where $0 \leq d_1 < d_2 \leq 1$.

The reason is that if $d(x, y) \leq d_1$, where d is the Jaccard distance, then $\text{SIM}(x, y) = 1 - d(x, y) \geq 1 - d_1$. But we know that the Jaccard similarity of x and y is equal to the probability that a minhash function will hash x and y to the same value. A similar argument applies to d_2 or any distance.

Example 3.18: We could let $d_1 = 0.3$ and $d_2 = 0.6$. Then we can assert that the family of minhash functions is a $(0.3, 0.6, 0.7, 0.4)$-sensitive family. That is, if the Jaccard distance between x and y is at most 0.3 (i.e., $\text{SIM}(x, y) \geq 0.7$) then there is at least a 0.7 chance that a minhash function will send x and y to the same value, and if the Jaccard distance between x and y is at least 0.6 (i.e., $\text{SIM}(x, y) \leq 0.4$), then there is at most a 0.4 chance that x and y will be sent to the same value. Note that we could make the same assertion with another choice of d_1 and d_2; only $d_1 < d_2$ is required. □

3.6.3 Amplifying a Locality-Sensitive Family

Suppose we are given a (d_1, d_2, p_1, p_2)-sensitive family $\mathbf{F}$. We can construct a new family $\mathbf{F}'$ by the *AND-construction* on $\mathbf{F}$, which is defined as follows. Each member of $\mathbf{F}'$ consists of r members of $\mathbf{F}$ for some fixed r. If f is in $\mathbf{F}'$, and f is constructed from the set $\{f_1, f_2, \ldots, f_r\}$ of members of $\mathbf{F}$, we say $f(x) = f(y)$ if and only if $f_i(x) = f_i(y)$ for all $i = 1, 2, \ldots, r$. Notice that this construction mirrors the effect of the r rows in a single band: the band makes x and y a

candidate pair if every one of the r rows in the band say that x and y are equal (and therefore a candidate pair according to that row).

Since the members of $\mathbf{F}$ are independently chosen to make a member of $\mathbf{F}'$, we can assert that $\mathbf{F}'$ is a $\big(d_1, d_2, (p_1)^r, (p_2)^r\big)$-sensitive family. That is, for any p, if p is the probability that a member of $\mathbf{F}$ will declare (x, y) to be a candidate pair, then the probability that a member of $\mathbf{F}'$ will so declare is p^r.

There is another construction, which we call the *OR-construction*, that turns a (d_1, d_2, p_1, p_2)-sensitive family $\mathbf{F}$ into a $\big(d_1, d_2, 1-(1-p_1)^b, 1-(1-p_2)^b\big)$-sensitive family $\mathbf{F}'$. Each member f of $\mathbf{F}'$ is constructed from b members of $\mathbf{F}$, say $f_1, f_2, \ldots, f_b$. We define $f(x) = f(y)$ if and only if $f_i(x) = f_i(y)$ for one or more values of i. The OR-construction mirrors the effect of combining several bands: x and y become a candidate pair if any band makes them a candidate pair.

If p is the probability that a member of $\mathbf{F}$ will declare (x, y) to be a candidate pair, then $1-p$ is the probability it will not so declare. $(1-p)^b$ is the probability that none of $f_1, f_2, \ldots, f_b$ will declare (x, y) a candidate pair, and $1-(1-p)^b$ is the probability that at least one f_i will declare (x, y) a candidate pair, and therefore that f will declare (x, y) to be a candidate pair.

Notice that the AND-construction lowers all probabilities, but if we choose $\mathbf{F}$ and r judiciously, we can make the small probability p_2 get very close to 0, while the higher probability p_1 stays significantly away from 0. Similarly, the OR-construction makes all probabilities rise, but by choosing $\mathbf{F}$ and b judiciously, we can make the larger probability approach 1 while the smaller probability remains bounded away from 1. We can cascade AND- and OR-constructions in any order to make the low probability close to 0 and the high probability close to 1. Of course the more constructions we use, and the higher the values of r and b that we pick, the larger the number of functions from the original family that we are forced to use. Thus, the better the final family of functions is, the longer it takes to apply the functions from this family.

Example 3.19: Suppose we start with a family $\mathbf{F}$. We use the AND-construction with $r = 4$ to produce a family $\mathbf{F}_1$. We then apply the OR-construction to $\mathbf{F}_1$ with $b = 4$ to produce a third family $\mathbf{F}_2$. Note that the members of $\mathbf{F}_2$ each are built from 16 members of $\mathbf{F}$, and the situation is analogous to starting with 16 minhash functions and treating them as four bands of four rows each.

The 4-way AND-function converts any probability p into p^4. When we follow it by the 4-way OR-construction, that probability is further converted into $1-(1-p^4)^4$. Some values of this transformation are indicated in Fig. 3.11. This function is an S-curve, staying low for a while, then rising steeply (although not too steeply; the slope never gets much higher than 2), and then leveling off at high values. Like any S-curve, it has a *fixedpoint*, the value of p that is left unchanged when we apply the function of the S-curve. In this case, the fixedpoint is the value of p for which $p = 1-(1-p^4)^4$. We can see that the fixedpoint is somewhere between 0.7 and 0.8. Below that value, probabilities are decreased, and above it they are increased. Thus, if we pick a high probability

p	$1-(1-p^4)^4$
0.2	0.0064
0.3	0.0320
0.4	0.0985
0.5	0.2275
0.6	0.4260
0.7	0.6666
0.8	0.8785
0.9	0.9860

Figure 3.11: Effect of the 4-way AND-construction followed by the 4-way OR-construction

above the fixedpoint and a low probability below it, we shall have the desired effect that the low probability is decreased and the high probability is increased.

Suppose $\mathbf{F}$ is the minhash functions, regarded as a $(0.2, 0.6, 0.8, 0.4)$-sensitive family. Then $\mathbf{F}_2$, the family constructed by a 4-way AND followed by a 4-way OR, is a $(0.2, 0.6, 0.8785, 0.0985)$-sensitive family, as we can read from the rows for 0.8 and 0.4 in Fig. 3.11. By replacing $\mathbf{F}$ by $\mathbf{F}_2$, we have reduced both the false-negative and false-positive rates, at the cost of making application of the functions take 16 times as long. □

p	$\left(1-(1-p)^4\right)^4$
0.1	0.0140
0.2	0.1215
0.3	0.3334
0.4	0.5740
0.5	0.7725
0.6	0.9015
0.7	0.9680
0.8	0.9936

Figure 3.12: Effect of the 4-way OR-construction followed by the 4-way AND-construction

Example 3.20 : For the same cost, we can apply a 4-way OR-construction followed by a 4-way AND-construction. Figure 3.12 gives the transformation on probabilities implied by this construction. For instance, suppose that $\mathbf{F}$ is a $(0.2, 0.6, 0.8, 0.4)$-sensitive family. Then the constructed family is a

$$(0.2, 0.6, 0.9936, 0.5740)\text{-sensitive}$$

family. This choice is not necessarily the best. Although the higher probability has moved much closer to 1, the lower probability has also raised, increasing the number of false positives. □

Example 3.21: We can cascade constructions as much as we like. For example, we could use the construction of Example 3.19 on the family of minhash functions and then use the construction of Example 3.20 on the resulting family. The constructed family would then have functions each built from 256 minhash functions. It would, for instance transform a $(0.2, 0.8, 0.8, 0.2)$-sensitive family into a $(0.2, 0.8, 0.9991285, 0.0000004)$-sensitive family. □

3.6.4 Exercises for Section 3.6

Exercise 3.6.1: What is the effect on probability of starting with the family of minhash functions and applying:

(a) A 2-way AND construction followed by a 3-way OR construction.

(b) A 3-way OR construction followed by a 2-way AND construction.

(c) A 2-way AND construction followed by a 2-way OR construction, followed by a 2-way AND construction.

(d) A 2-way OR construction followed by a 2-way AND construction, followed by a 2-way OR construction followed by a 2-way AND construction.

Exercise 3.6.2: Find the fixedpoints for each of the functions constructed in Exercise 3.6.1.

! **Exercise 3.6.3:** Any function of probability p, such as that of Fig. 3.11, has a slope given by the derivative of the function. The maximum slope is where that derivative is a maximum. Find the value of p that gives a maximum slope for the S-curves given by Fig. 3.11 and Fig. 3.12. What are the values of these maximum slopes?

!! **Exercise 3.6.4:** Generalize Exercise 3.6.3 to give, as a function of r and b, the point of maximum slope and the value of that slope, for families of functions defined from the minhash functions by:

(a) An r-way AND construction followed by a b-way OR construction.

(b) A b-way OR construction followed by an r-way AND construction.

3.7 LSH Families for Other Distance Measures

There is no guarantee that a distance measure has a locality-sensitive family of hash functions. So far, we have only seen such families for the Jaccard distance. In this section, we shall show how to construct locality-sensitive families for Hamming distance, the cosine distance and for the normal Euclidean distance.

3.7.1 LSH Families for Hamming Distance

It is quite simple to build a locality-sensitive family of functions for the Hamming distance. Suppose we have a space of d-dimensional vectors, and $h(x, y)$ denotes the Hamming distance between vectors x and y. If we take any one position of the vectors, say the ith position, we can define the function $f_i(x)$ to be the ith bit of vector x. Then $f_i(x) = f_i(y)$ if and only if vectors x and y agree in the ith position. Then the probability that $f_i(x) = f_i(y)$ for a randomly chosen i is exactly $1 - h(x, y)/d$; i.e., it is the fraction of positions in which x and y agree.

This situation is almost exactly like the one we encountered for minhashing. Thus, the family $\mathbf{F}$ consisting of the functions $\{f_1, f_2, \ldots, f_d\}$ is a

$$(d_1, d_2, 1 - d_1/d, 1 - d_2/d)\text{-sensitive}$$

family of hash functions, for any $d_1 < d_2$. There are only two differences between this family and the family of minhash functions.

1. While Jaccard distance runs from 0 to 1, the Hamming distance on a vector space of dimension d runs from 0 to d. It is therefore necessary to scale the distances by dividing by d, to turn them into probabilities.

2. While there is essentially an unlimited supply of minhash functions, the size of the family $\mathbf{F}$ for Hamming distance is only d.

The first point is of no consequence; it only requires that we divide by d at appropriate times. The second point is more serious. If d is relatively small, then we are limited in the number of functions that can be composed using the AND and OR constructions, thereby limiting how steep we can make the S-curve be.

3.7.2 Random Hyperplanes and the Cosine Distance

Recall from Section 3.5.4 that the cosine distance between two vectors is the angle between the vectors. For instance, we see in Fig. 3.13 two vectors x and y that make an angle θ between them. Note that these vectors may be in a space of many dimensions, but they always define a plane, and the angle between them is measured in this plane. Figure 3.13 is a "top-view" of the plane containing x and y.

Suppose we pick a hyperplane through the origin. This hyperplane intersects the plane of x and y in a line. Figure 3.13 suggests two possible hyperplanes, one whose intersection is the dashed line and the other's intersection is the dotted line. To pick a random hyperplane, we actually pick the normal vector to the hyperplane, say v. The hyperplane is then the set of points whose dot product with v is 0.

First, consider a vector v that is normal to the hyperplane whose projection is represented by the dashed line in Fig. 3.13; that is, x and y are on different

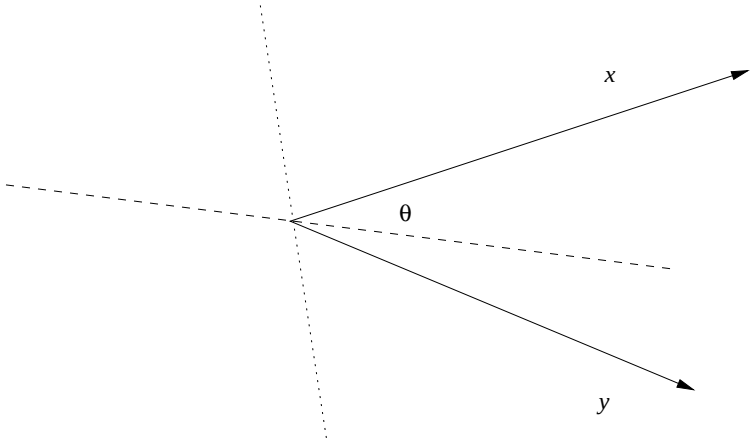

Figure 3.13: Two vectors make an angle θ

sides of the hyperplane. Then the dot products $v.x$ and $v.y$ will have different signs. If we assume, for instance, that v is a vector whose projection onto the plane of x and y is above the dashed line in Fig. 3.13, then $v.x$ is positive, while $v.y$ is negative. The normal vector v instead might extend in the opposite direction, below the dashed line. In that case $v.x$ is negative and $v.y$ is positive, but the signs are still different.

On the other hand, the randomly chosen vector v could be normal to a hyperplane like the dotted line in Fig. 3.13. In that case, both $v.x$ and $v.y$ have the same sign. If the projection of v extends to the right, then both dot products are positive, while if v extends to the left, then both are negative.

What is the probability that the randomly chosen vector is normal to a hyperplane that looks like the dashed line rather than the dotted line? All angles for the line that is the intersection of the random hyperplane and the plane of x and y are equally likely. Thus, the hyperplane will look like the dashed line with probability $\theta/180$ and will look like the dotted line otherwise.

Thus, each hash function f in our locality-sensitive family $\mathbf{F}$ is built from a randomly chosen vector v_f. Given two vectors x and y, say $f(x) = f(y)$ if and only if the dot products $v_f.x$ and $v_f.y$ have the same sign. Then $\mathbf{F}$ is a locality-sensitive family for the cosine distance. The parameters are essentially the same as for the Jaccard-distance family described in Section 3.6.2, except the scale of distances is 0–180 rather than 0–1. That is, $\mathbf{F}$ is a

$$(d_1, d_2, (180 - d_1)/180, (180 - d_2)/180)\text{-sensitive}$$

family of hash functions. From this basis, we can amplify the family as we wish, just as for the minhash-based family.

3.7.3 Sketches

Instead of chosing a random vector from all possible vectors, it turns out to be sufficiently random if we restrict our choice to vectors whose components are $+1$ and -1. The dot product of any vector x with a vector v of $+1$'s and -1's is formed by adding the components of x where v is $+1$ and then subtracting the other components of x – those where v is -1.

If we pick a collection of random vectors, say $v_1, v_2, \ldots, v_n$, then we can apply them to an arbitrary vector x by computing $v_1.x, v_2.x, \ldots, v_n.x$ and then replacing any positive value by $+1$ and any negative value by -1. The result is called the *sketch* of x. You can handle 0's arbitrarily, e.g., by chosing a result $+1$ or -1 at random. Since there is only a tiny probability of a zero dot product, the choice has essentially no effect.

Example 3.22: Suppose our space consists of 4-dimensional vectors, and we pick three random vectors: $v_1 = [+1, -1, +1, +1]$, $v_2 = [-1, +1, -1, +1]$, and $v_3 = [+1, +1, -1, -1]$. For the vector $x = [3, 4, 5, 6]$, the sketch is $[+1, +1, -1]$. That is, $v_1.x = 3-4+5+6 = 10$. Since the result is positive, the first component of the sketch is $+1$. Similarly, $v_2.x = 2$ and $v_3.x = -4$, so the second component of the sketch is $+1$ and the third component is -1.

Consider the vector $y = [4, 3, 2, 1]$. We can similarly compute its sketch to be $[+1, -1, +1]$. Since the sketches for x and y agree in 1/3 of the positions, we estimate that the angle between them is 120 degrees. That is, a randomly chosen hyperplane is twice as likely to look like the dashed line in Fig. 3.13 than like the dotted line.

The above conclusion turns out to be quite wrong. We can calculate the cosine of the angle between x and y to be $x.y$, which is

$$6 \times 1 + 5 \times 2 + 4 \times 3 + 3 \times 4 = 40$$

divided by the magnitudes of the two vectors. These magnitudes are

$$\sqrt{6^2 + 5^2 + 4^2 + 3^2} = 9.274$$

and $\sqrt{1^2 + 2^2 + 3^2 + 4^2} = 5.477$. Thus, the cosine of the angle between x and y is 0.7875, and this angle is about 38 degrees. However, if you look at all 16 different vectors v of length 4 that have $+1$ and -1 as components, you find that there are only four of these whose dot products with x and y have a different sign, namely v_2, v_3, and their complements $[+1, -1, +1, -1]$ and $[-1, -1, +1, +1]$. Thus, had we picked all sixteen of these vectors to form a sketch, the estimate of the angle would have been $180/4 = 45$ degrees. □

3.7.4 LSH Families for Euclidean Distance

Now, let us turn to the Euclidean distance (Section 3.5.2), and see if we can develop a locality-sensitive family of hash functions for this distance. We shall start with a 2-dimensional Euclidean space. Each hash function f in our family

F will be associated with a randomly chosen line in this space. Pick a constant a and divide the line into segments of length a, as suggested by Fig. 3.14, where the "random" line has been oriented to be horizontal.

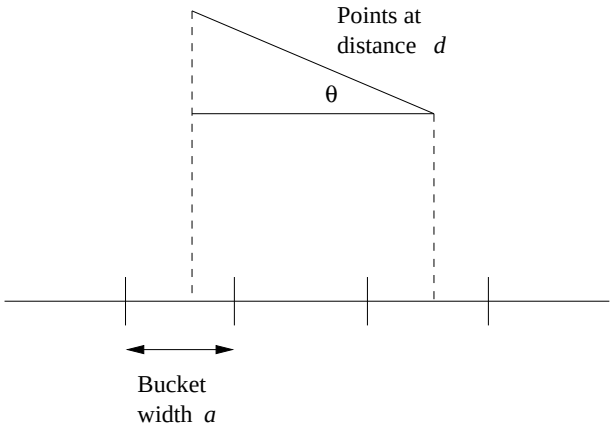

Figure 3.14: Two points at distance $d \gg a$ have a small chance of being hashed to the same bucket

The segments of the line are the buckets into which function f hashes points. A point is hashed to the bucket in which its projection onto the line lies. If the distance d between two points is small compared with a, then there is a good chance the two points hash to the same bucket, and thus the hash function f will declare the two points equal. For example, if $d = a/2$, then there is at least a 50% chance the two points will fall in the same bucket. In fact, if the angle θ between the randomly chosen line and the line connecting the points is large, then there is an even greater chance that the two points will fall in the same bucket. For instance, if θ is 90 degrees, then the two points are certain to fall in the same bucket.

However, suppose d is larger than a. In order for there to be any chance of the two points falling in the same bucket, we need $d \cos\theta \leq a$. The diagram of Fig. 3.14 suggests why this requirement holds. Note that even if $d \cos\theta \ll a$ it is still not certain that the two points will fall in the same bucket. However, we can guarantee the following. If $d \geq 2a$, then there is no more than a 1/3 chance the two points fall in the same bucket. The reason is that for $\cos\theta$ to be less than 1/2, we need to have θ in the range 60 to 90 degrees. If θ is in the range 0 to 60 degrees, then $\cos\theta$ is more than 1/2. But since θ is the smaller angle between two randomly chosen lines in the plane, θ is twice as likely to be between 0 and 60 as it is to be between 60 and 90.

We conclude that the family **F** just described forms a $(a/2, 2a, 1/2, 1/3)$-sensitive family of hash functions. That is, for distances up to $a/2$ the probability is at least 1/2 that two points at that distance will fall in the same bucket, while for distances at least $2a$ the probability points at that distance will fall in

the same bucket is at most 1/3. We can amplify this family as we like, just as for the other examples of locality-sensitive hash functions we have discussed.

3.7.5 More LSH Families for Euclidean Spaces

There is something unsatisfying about the family of hash functions developed in Section 3.7.4. First, the technique was only described for two-dimensional Euclidean spaces. What happens if our data is points in a space with many dimensions? Second, for Jaccard and cosine distances, we were able to develop locality-sensitive families for any pair of distances d_1 and d_2 as long as $d_1 < d_2$. In Section 3.7.4 we appear to need the stronger condition $d_1 < 4d_2$.

However, we claim that there is a locality-sensitive family of hash functions for any $d_1 < d_2$ and for any number of dimensions. The family's hash functions still derive from random lines through the space and a bucket size a that partitions the line. We still hash points by projecting them onto the line. Given that $d_1 < d_2$, we may not know what the probability p_1 is that two points at distance d_1 hash to the same bucket, but we can be certain that it is greater than p_2, the probability that two points at distance d_2 hash to the same bucket. The reason is that this probability surely grows as the distance shrinks. Thus, even if we cannot calculate p_1 and p_2 easily, we know that there is a (d_1, d_2, p_1, p_2)-sensitive family of hash functions for any $d_1 < d_2$ and any given number of dimensions.

Using the amplification techniques of Section 3.6.3, we can then adjust the two probabilities to surround any particular value we like, and to be as far apart as we like. Of course, the further apart we want the probabilities to be, the larger the number of basic hash functions in $\mathbf{F}$ we must use.

3.7.6 Exercises for Section 3.7

Exercise 3.7.1: Suppose we construct the basic family of six locality-sensitive functions for vectors of length six. For each pair of the vectors 000000, 110011, 010101, and 011100, which of the six functions makes them candidates?

Exercise 3.7.2: Let us compute sketches using the following four "random" vectors:

$$v_1 = [+1, +1, +1, -1] \quad v_2 = [+1, +1, -1, +1]$$
$$v_3 = [+1, -1, +1, +1] \quad v_4 = [-1, +1, +1, +1]$$

Compute the sketches of the following vectors.

(a) $[2, 3, 4, 5]$.

(b) $[-2, 3, -4, 5]$.

(c) $[2, -3, 4, -5]$.

For each pair, what is the estimated angle between them, according to the sketches? What are the true angles?

Exercise 3.7.3: Suppose we form sketches by using all sixteen of the vectors of length 4, whose components are each $+1$ or -1. Compute the sketches of the three vectors in Exercise 3.7.2. How do the estimates of the angles between each pair compare with the true angles?

Exercise 3.7.4: Suppose we form sketches using the four vectors from Exercise 3.7.2.

! (a) What are the constraints on a, b, c, and d that will cause the sketch of the vector $[a, b, c, d]$ to be $[+1, +1, +1, +1]$?

!! (b) Consider two vectors $[a, b, c, d]$ and $[e, f, g, h]$. What are the conditions on $a, b, \ldots, h$ that will make the sketches of these two vectors be the same?

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 per-

son, 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 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

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.

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 3.23: 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 3.23 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 $\mathbf{F}$ 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 $\mathbf{F}$ 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 3.24: 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 3.24 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 3.24 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.[7] 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 3.23, 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?

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

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 3.23, 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?

3.9 Methods for High Degrees of Similarity

LSH-based methods appear most effective when the degree of similarity we accept is relatively low. When we want to find sets that are almost identical, there are other methods that can be faster. Moreover, these methods are exact, in that they find every pair of items with the desired degree of similarity. There are no false negatives, as there can be with LSH.

3.9.1 Finding Identical Items

The extreme case is finding identical items, for example, Web pages that are identical, character-for-character. It is straightforward to compare two documents and tell whether they are identical, but we still must avoid having to compare every pair of documents. Our first thought would be to hash documents based on their first few characters, and compare only those documents that fell into the same bucket. That scheme should work well, unless all the documents begin with the same characters, such as an HTML header.

Our second thought would be to use a hash function that examines the entire document. That would work, and if we use enough buckets, it would be very rare that two documents went into the same bucket, yet were not identical. The downside of this approach is that we must examine every character of every document. If we limit our examination to a small number of characters, then we never have to examine a document that is unique and falls into a bucket of its own.

A better approach is to pick some fixed random positions for all documents, and make the hash function depend only on these. This way, we can avoid

a problem where there is a common prefix for all or most documents, yet we need not examine entire documents unless they fall into a bucket with another document. One problem with selecting fixed positions is that if some documents are short, they may not have some of the selected positions. However, if we are looking for highly similar documents, we never need to compare two documents that differ significantly in their length. We exploit this idea in Section 3.9.3.

3.9.2 Representing Sets as Strings

Now, let us focus on the harder problem of finding, in a large collection of sets, all pairs that have a high Jaccard similarity, say at least 0.9. We can represent a set by sorting the elements of the universal set in some fixed order, and representing any set by listing its elements in this order. The list is essentially a string of "characters," where the characters are the elements of the universal set. These strings are unusual, however, in that:

1. No character appears more than once in a string, and
2. If two characters appear in two different strings, then they appear in the same order in both strings.

Example 3.25: Suppose the universal set consists of the 26 lower-case letters, and we use the normal alphabetical order. Then the set $\{d, a, b\}$ is represented by the string `abd`. □

In what follows, we shall assume all strings represent sets in the manner just described. Thus, we shall talk about the Jaccard similarity of strings, when strictly speaking we mean the similarity of the sets that the strings represent. Also, we shall talk of the length of a string, as a surrogate for the number of elements in the set that the string represents.

Note that the documents discussed in Section 3.9.1 do not exactly match this model, even though we can see documents as strings. To fit the model, we would shingle the documents, assign an order to the shingles, and represent each document by its list of shingles in the selected order.

3.9.3 Length-Based Filtering

The simplest way to exploit the string representation of Section 3.9.2 is to sort the strings by length. Then, each string s is compared with those strings t that follow s in the list, but are not too long. Suppose the lower bound on Jaccard similarity between two strings is J. For any string x, denote its length by L_x. Note that $L_s \leq L_t$. The intersection of the sets represented by s and t cannot have more than L_s members, while their union has at least L_t members. Thus, the Jaccard similarity of s and t, which we denote $\text{SIM}(s, t)$, is at most L_s/L_t. That is, in order for s and t to require comparison, it must be that $J \leq L_s/L_t$, or equivalently, $L_t \leq L_s/J$.

A Better Ordering for Symbols

Instead of using the obvious order for elements of the universal set, e.g., lexicographic order for shingles, we can order symbols rarest first. That is, determine how many times each element appears in the collection of sets, and order them by this count, lowest first. The advantage of doing so is that the symbols in prefixes will tend to be rare. Thus, they will cause that string to be placed in index buckets that have relatively few members. Then, when we need to examine a string for possible matches, we shall find few other strings that are candidates for comparison.

Example 3.26 : Suppose that s is a string of length 9, and we are looking for strings with at least 0.9 Jaccard similarity. Then we have only to compare s with strings following it in the length-based sorted order that have length at most $9/0.9 = 10$. That is, we compare s with those strings of length 9 that follow it in order, and all strings of length 10. We have no need to compare s with any other string.

Suppose the length of s were 8 instead. Then s would be compared with following strings of length up to $8/0.9 = 8.89$. That is, a string of length 9 would be too long to have a Jaccard similarity of 0.9 with s, so we only have to compare s with the strings that have length 8 but follow it in the sorted order. □

3.9.4 Prefix Indexing

In addition to length, there are several other features of strings that can be exploited to limit the number of comparisons that must be made to identify all pairs of similar strings. The simplest of these options is to create an index for each symbol; recall a symbol of a string is any one of the elements of the universal set. For each string s, we select a prefix of s consisting of the first p symbols of s. How large p must be depends on L_s and J, the lower bound on Jaccard similarity. We add string s to the index for each of its first p symbols.

In effect, the index for each symbol becomes a bucket of strings that must be compared. We must be certain that any other string t such that $\text{SIM}(s,t) \geq J$ will have at least one symbol in its prefix that also appears in the prefix of s.

Suppose not; rather $\text{SIM}(s,t) \geq J$, but t has none of the first p symbols of s. Then the highest Jaccard similarity that s and t can have occurs when t is a suffix of s, consisting of everything but the first p symbols of s. The Jaccard similarity of s and t would then be $(L_s - p)/L_s$. To be sure that we do not have to compare s with t, we must be certain that $J > (L_s - p)/L_s$. That is, p must be at least $\lfloor (1-J)L_s \rfloor + 1$. Of course we want p to be as small as possible, so we do not index string s in more buckets than we need to. Thus, we shall hereafter take $p = \lfloor (1-J)L_s \rfloor + 1$ to be the length of the prefix that

gets indexed.

Example 3.27: Suppose $J = 0.9$. If $L_s = 9$, then $p = \lfloor 0.1 \times 9 \rfloor + 1 = \lfloor 0.9 \rfloor + 1 = 1$. That is, we need to index s under only its first symbol. Any string t that does not have the first symbol of s in a position such that t is indexed by that symbol will have Jaccard similarity with s that is less than 0.9. Suppose s is `bcdefghij`. Then s is indexed under `b` only. Suppose t does not begin with `b`. There are two cases to consider.

1. If t begins with `a`, and $\text{SIM}(s,t) \geq 0.9$, then it can only be that t is `abcdefghij`. But if that is the case, t will be indexed under both `a` and `b`. The reason is that $L_t = 10$, so t will be indexed under the symbols of its prefix of length $\lfloor 0.1 \times 10 \rfloor + 1 = 2$.

2. If t begins with `c` or a later letter, then the maximum value of $\text{SIM}(s,t)$ occurs when t is `cdefghij`. But then $\text{SIM}(s,t) = 8/9 < 0.9$.

In general, with $J = 0.9$, strings of length up to 9 are indexed by their first symbol, strings of lengths 10–19 are indexed under their first two symbols, strings of length 20–29 are indexed under their first three symbols, and so on. □

We can use the indexing scheme in two ways, depending on whether we are trying to solve the many-many problem or a many-one problem; recall the distinction was introduced in Section 3.8.4. For the many-one problem, we create the index for the entire database. To query for matches to a new set S, we convert that set to a string s, which we call the *probe* string. Determine the length of the prefix that must be considered, that is, $\lfloor (1-J)L_s \rfloor + 1$. For each symbol appearing in one of the prefix positions of s, we look in the index bucket for that symbol, and we compare s with all the strings appearing in that bucket.

If we want to solve the many-many problem, start with an empty database of strings and indexes. For each set S, we treat S as a new set for the many-one problem. We convert S to a string s, which we treat as a probe string in the many-one problem. However, after we examine an index bucket, we also add s to that bucket, so s will be compared with later strings that could be matches.

3.9.5 Using Position Information

Consider the strings $s =$ `acdefghijk` and $t =$ `bcdefghijk`, and assume $J = 0.9$. Since both strings are of length 10, they are indexed under their first two symbols. Thus, s is indexed under `a` and `c`, while t is indexed under `b` and `c`. Whichever is added last will find the other in the bucket for `c`, and they will be compared. However, since `c` is the second symbol of both, we know there will be two symbols, `a` and `b` in this case, that are in the union of the two sets but not in the intersection. Indeed, even though s and t are identical from `c` to the

end, their intersection is 9 symbols and their union is 11; thus SIM$(s,t) = 9/11$, which is less than 0.9.

If we build our index based not only on the symbol, but on the position of the symbol within the string, we could avoid comparing s and t above. That is, let our index have a bucket for each pair (x,i), containing the strings that have symbol x in position i of their prefix. Given a string s, and assuming J is the minimum desired Jaccard similarity, we look at the prefix of s, that is, the positions 1 through $\lfloor(1-J)L_s\rfloor + 1$. If the symbol in position i of the prefix is x, add s to the index bucket for (x,i).

Now consider s as a probe string. With what buckets must it be compared? We shall visit the symbols of the prefix of s from the left, and we shall take advantage of the fact that we only need to find a possible matching string t if none of the previous buckets we have examined for matches held t. That is, we only need to find a candidate match once. Thus, if we find that the ith symbol of s is x, then we need look in the bucket (x,j) for certain small values of j.

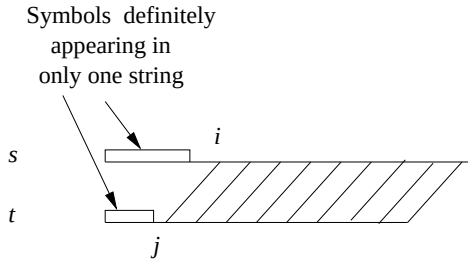

Figure 3.15: Strings s and t begin with $i-1$ and $j-1$ unique symbols, respectively, and then agree beyond that

To compute the upper bound on j, suppose t is a string none of whose first $j-1$ symbols matched anything in s, but the ith symbol of s is the same as the jth symbol of t. The highest value of SIM(s,t) occurs if s and t are identical beyond their ith and jth symbols, respectively, as suggested by Fig. 3.15. If that is the case, the size of their intersection is $L_s - i + 1$, since that is the number of symbols of s that could possibly be in t. The size of their union is at least $L_s + j - 1$. That is, s surely contributes L_s symbols to the union, and there are also at least $j-1$ symbols of t that are not in s. The ratio of the sizes of the intersection and union must be at least J, so we must have:

$$\frac{L_s - i + 1}{L_s + j - 1} \geq J$$

If we isolate j in this inequality, we have $j \leq \big(L_s(1-J) - i + 1 + J\big)/J$.

Example 3.28: Consider the string $s =$ `acdefghijk` with $J = 0.9$ discussed at the beginning of this section. Suppose s is now a probe string. We already established that we need to consider the first two positions; that is, i can be 1

or 2. Suppose $i = 1$. Then $j \leq (10 \times 0.1 - 1 + 1 + 0.9)/0.9$. That is, we only have to compare the symbol `a` with strings in the bucket for $(\texttt{a}, j)$ if $j \leq 2.11$. Thus, j can be 1 or 2, but nothing higher.

Now suppose $i = 2$. Then we require $j \leq (10 \times 0.1 - 2 + 1 + 0.9)/0.9$, Or $j \leq 1$. We conclude that we must look in the buckets for $(\texttt{a}, 1)$, $(\texttt{a}, 2)$, and $(\texttt{c}, 1)$, but in no other bucket. In comparison, using the buckets of Section 3.9.4, we would look into the buckets for `a` and `c`, which is equivalent to looking to all buckets $(\texttt{a}, j)$ and $(\texttt{c}, j)$ for any j. □

3.9.6 Using Position and Length in Indexes

When we considered the upper limit on j in the previous section, we assumed that what follows positions i and j were as in Fig. 3.15, where what followed these positions in strings s and t matched exactly. We do not want to build an index that involves every symbol in the strings, because that makes the total work excessive. However, we can add to our index a summary of what follows the positions being indexed. Doing so expands the number of buckets, but not beyond reasonable bounds, and yet enables us to eliminate many candidate matches without comparing entire strings. The idea is to use index buckets corresponding to a symbol, a position, and the *suffix length*, that is, the number of symbols following the position in question.

Example 3.29: The string $s = \texttt{acdefghijk}$, with $J = 0.9$, would be indexed in the buckets for $(\texttt{a}, 1, 9)$ and $(\texttt{c}, 2, 8)$. That is, the first position of s has symbol `a`, and its suffix is of length 9. The second position has symbol `c` and its suffix is of length 8. □

Figure 3.15 assumes that the suffixes for position i of s and position j of t have the same length. If not, then we can either get a smaller upper bound on the size of the intersection of s and t (if t is shorter) or a larger lower bound on the size of the union (if t is longer). Suppose s has suffix length p and t has suffix length q.

Case 1: $p \geq q$**.** Here, the maximum size of the intersection is

$$L_s - i + 1 - (p - q)$$

Since $L_s = i + p$, we can write the above expression for the intersection size as $q + 1$. The minimum size of the union is $L_s + j - 1$, as it was when we did not take suffix length into account. Thus, we require

$$\frac{q+1}{L_s + j - 1} \geq J$$

whenever $p \geq q$.

Case 2: $p < q$. Here, the maximum size of the intersection is $L_s - i + 1$, as when suffix length was not considered. However, the minimum size of the union is now $L_s + j - 1 + q - p$. If we again use the relationship $L_s = i + p$, we can replace $L_s - p$ by i and get the formula $i + j - 1 + q$ for the size of the union. If the Jaccard similarity is at least J, then

$$\frac{L_s - i + 1}{i + j - 1 + q} \geq J$$

whenever $p < q$.

Example 3.30: Let us again consider the string $s =$ `acdefghijk`, but to make the example show some details, let us choose $J = 0.8$ instead of 0.9. We know that $L_s = 10$. Since $\lfloor (1 - J)L_s \rfloor + 1 = 3$, we must consider prefix positions $i = 1$, 2, and 3 in what follows. As before, let p be the suffix length of s and q the suffix length of t.

First, consider the case $p \geq q$. The additional constraint we have on q and j is $(q + 1)/(9 + j) \geq 0.8$. We can enumerate the pairs of values of j and q for each i between 1 and 3, as follows.

$i = 1$**:** Here, $p = 9$, so $q \leq 9$. Let us consider the possible values of q:

- $q = 9$**:** We must have $10/(9 + j) \geq 0.8$. Thus, we can have $j = 1$, $j = 2$, or $j = 3$. Note that for $j = 4$, $10/13 > 0.8$.
- $q = 8$**:** We must have $9/(9 + j) \geq 0.8$. Thus, we can have $j = 1$ or $j = 2$. For $j = 3$, $9/12 > 0.8$.
- $q = 7$**:** We must have $8/(9 + j) \geq 0.8$. Only $j = 1$ satisfies this inequality.
- $q = 6$**:** There are no possible values of j, since $7/(9 + j) > 0.8$ for every positive integer j. The same holds for every smaller value of q.

$i = 2$**:** Here, $p = 8$, so we require $q \leq 8$. Since the constraint $(q+1)/(9+j) \geq 0.8$ does not depend on i,[8] we can use the analysis from the above case, but exclude the case $q = 9$. Thus, the only possible values of j and q when $i = 2$ are

1. $q = 8$; $j = 1$.
2. $q = 8$; $j = 2$.
3. $q = 7$; $j = 1$.

$i = 3$**:** Now, $p = 7$ and the constraints are $q \leq 7$ and $(q + 1)/(9 + j) \geq 0.8$. The only option is $q = 7$ and $j = 1$.

Next, we must consider the case $p < q$. The additional constraint is

$$\frac{11 - i}{i + j + q - 1} \geq 0.8$$

Again, consider each possible value of i.

[8] Note that i does influence the value of p, and through p, puts a limit on q.

$i = 1$**:** Then $p = 9$, so we require $q \geq 10$ and $10/(q + j) \geq 0.8$. The possible values of q and j are

1. $q = 10$; $j = 1$.
2. $q = 10$; $j = 2$.
3. $q = 11$; $j = 1$.

$i = 2$**:** Now, $p = 8$, so we require $q \geq 9$ and $9/(q + j + 1) \geq 0.8$. Since j must be a positive integer, the only solution is $q = 9$ and $j = 1$, a possibility that we already knew about.

$i = 3$**:** Here, $p = 7$, so we require $q \geq 8$ and $8/(q + j + 2) \geq 0.8$. There are no solutions.

	q	$j = 1$	$j = 2$	$j = 3$
$i = 1$	7	x		
	8	x	x	
	9	x	x	x
	10	x	x	
	11	x		
$i = 2$	7	x		
	8	x	x	
	9	x		
$i = 3$	7	x		

Figure 3.16: The buckets that must be examined to find possible matches for the string $s =$ `acdefghijk` with $J = 0.8$ are marked with an x

When we accumulate the possible combinations of i, j, and q, we see that the set of index buckets in which we must look forms a pyramid. Figure 3.16 shows the buckets in which we must search. That is, we must look in those buckets (x, j, q) such that the ith symbol of the string s is x, j is the position associated with the bucket and q the suffix length. □

3.9.7 Exercises for Section 3.9

Exercise 3.9.1: Suppose our universal set is the lower-case letters, and the order of elements is taken to be the vowels, in alphabetic order, followed by the consonants in reverse alphabetic order. Represent the following sets as strings.

a $\{q, w, e, r, t, y\}$.

(b) $\{a, s, d, f, g, h, j, u, i\}$.

Exercise 3.9.2: Suppose we filter candidate pairs based only on length, as in Section 3.9.3. If s is a string of length 20, with what strings is s compared when J, the lower bound on Jaccard similarity has the following values: (a) $J = 0.85$ (b) $J = 0.95$ (c) $J = 0.98$?

Exercise 3.9.3: Suppose we have a string s of length 15, and we wish to index its prefix as in Section 3.9.4.

(a) How many positions are in the prefix if $J = 0.85$?

(b) How many positions are in the prefix if $J = 0.95$?

! (c) For what range of values of J will s be indexed under its first four symbols, but no more?

Exercise 3.9.4: Suppose s is a string of length 12. With what symbol-position pairs will s be compared with if we use the indexing approach of Section 3.9.5, and (a) $J = 0.75$ (b) $J = 0.95$?

! Exercise 3.9.5: Suppose we use position information in our index, as in Section 3.9.5. Strings s and t are both chosen at random from a universal set of 100 elements. Assume $J = 0.9$. What is the probability that s and t will be compared if

(a) s and t are both of length 9.

(b) s and t are both of length 10.

Exercise 3.9.6: Suppose we use indexes based on both position and suffix length, as in Section 3.9.6. If s is a string of length 20, with what symbol-position-length triples will s be compared with, if (a) $J = 0.8$ (b) $J = 0.9$?

3.10 Summary of Chapter 3

- ✦ *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 permutations 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. An additional efficiency can be had by restricting the search for the smallest minhash value to only a small subset of the universal set.

- *Locality-Sensitive Hashing for Signatures*: This technique allows us to avoid computing the similarity of every pair of sets or their minhash signatures. 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 identical 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 different. 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.

- *Euclidean Distance*: The most common notion of distance is the Euclidean distance in an n-dimensional space. This distance, sometimes called the L_2-norm, is the square root of the sum of the squares of the differences between the points in each dimension. Another distance suitable for Euclidean spaces, called Manhattan distance or the L_1-norm is the sum of the magnitudes of the differences between the points in each dimension.

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

- *Cosine Distance*: The angle between vectors in a vector space is the cosine distance measure. We can compute the cosine of that angle by taking the dot product of the vectors and dividing by the lengths of the vectors.

- *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.

- ✦ *Hamming Distance*: This distance measure applies to a space of vectors. The Hamming distance between two vectors is the number of positions in which the vectors differ.
- ✦ *Generalized Locality-Sensitive Hashing*: We may start with any collection of functions, such as the minhash functions, that can render a decision as to whether or not a pair of items should be candidates for similarity checking. The only constraint on these functions is that they provide a lower bound on the probability of saying "yes" if the distance (according to some distance measure) is below a given limit, and an upper bound on the probability of saying "yes" if the distance is above another given limit. We can then increase the probability of saying "yes" for nearby items and at the same time decrease the probability of saying "yes" for distant items to as great an extent as we wish, by applying an AND construction and an OR construction.
- ✦ *Random Hyperplanes and LSH for Cosine Distance*: We can get a set of basis functions to start a generalized LSH for the cosine distance measure by identifying each function with a list of randomly chosen vectors. We apply a function to a given vector v by taking the dot product of v with each vector on the list. The result is a sketch consisting of the signs ($+1$ or -1) of the dot products. The fraction of positions in which the sketches of two vectors agree, multiplied by 180, is an estimate of the angle between the two vectors.
- ✦ *LSH For Euclidean Distance*: A set of basis functions to start LSH for Euclidean distance can be obtained by choosing random lines and projecting points onto those lines. Each line is broken into fixed-length intervals, and the function answers "yes" to a pair of points that fall into the same interval.
- ✦ *High-Similarity Detection by String Comparison*: An alternative approach to finding similar items, when the threshold of Jaccard similarity is close to 1, avoids using minhashing and LSH. Rather, the universal set is ordered, and sets are represented by strings, consisting their elements in order. The simplest way to avoid comparing all pairs of sets or their strings is to note that highly similar sets will have strings of approximately the same length. If we sort the strings, we can compare each string with only a small number of the immediately following strings.
- ✦ *Character Indexes*: If we represent sets by strings, and the similarity threshold is close to 1, we can index all strings by their first few characters. The prefix whose characters must be indexed is approximately the length of the string times the maximum Jaccard distance (1 minus the minimum Jaccard similarity).

- *Position Indexes*: We can index strings not only on the characters in their prefixes, but on the position of that character within the prefix. We reduce the number of pairs of strings that must be compared, because if two strings share a character that is not in the first position in both strings, then we know that either there are some preceding characters that are in the union but not the intersection, or there is an earlier symbol that appears in both strings.

- *Suffix Indexes*: We can also index strings based not only on the characters in their prefixes and the positions of those characters, but on the length of the character's suffix – the number of positions that follow it in the string. This structure further reduces the number of pairs that must be compared, because a common symbol with different suffix lengths implies additional characters that must be in the union but not in the intersection.

3.11 References for Chapter 3

The technique we called shingling is attributed to [11]. The use in the manner we discussed here is from [2].

Minhashing comes from [3]. The improvement that avoids looking at all elements is from [10].

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]. [12] 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 [13].

1. A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate 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 algorithms," *ACM Symposium on Theory of Computing*, pp. 380–388, 2002.

5. S. Chaudhuri, V. Ganti, and R. Kaushik, "A primitive operator for similarity 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 Computational Geometry* pp. 253–262, 2004.

7. A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions 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 removing the curse of dimensionality," *ACM Symposium on Theory of Computing*, pp. 604–613, 1998.

10. P. Li, A.B. Owen, and C.H. Zhang. "One permutation hashing," *Conf. on Neural Information Processing Systems* 2012, pp. 3122–3130.

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

12. M. Theobald, J. Siddharth, and A. Paepcke, "SpotSigs: robust and efficient near duplicate detection in large web collections," *31st Annual ACM SIGIR Conference*, July, 2008, Singapore.

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