**Unit III: SHINGLING OF DOCUMENTS**

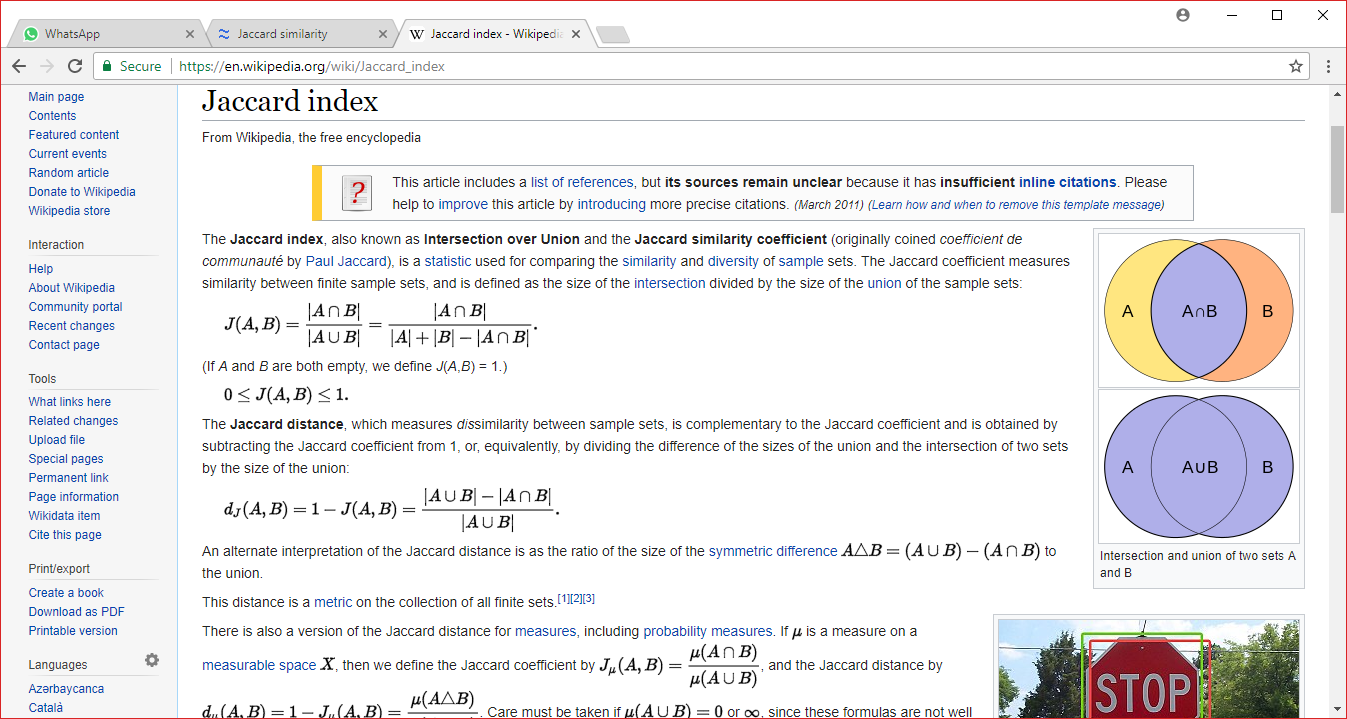
**Jaccard similarity**

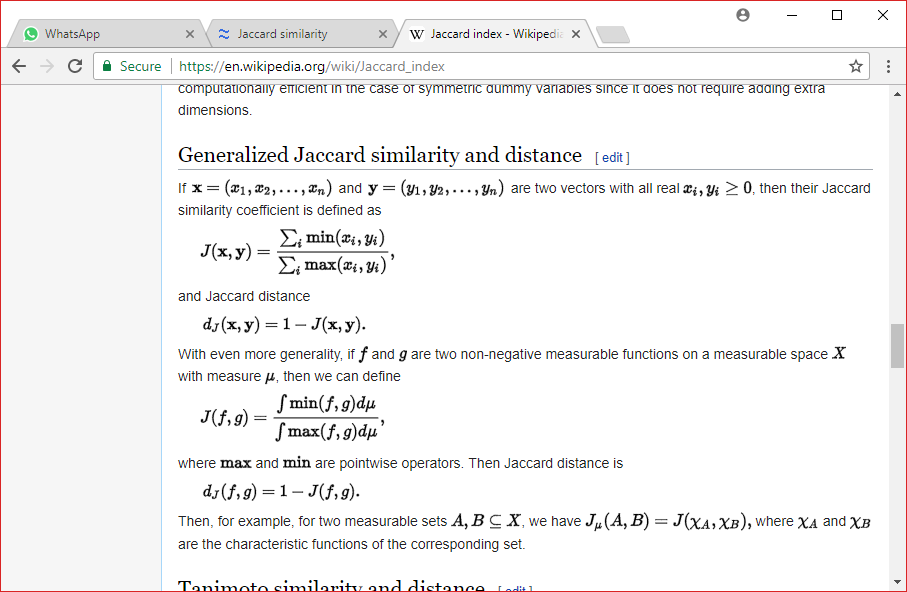
**Objective**

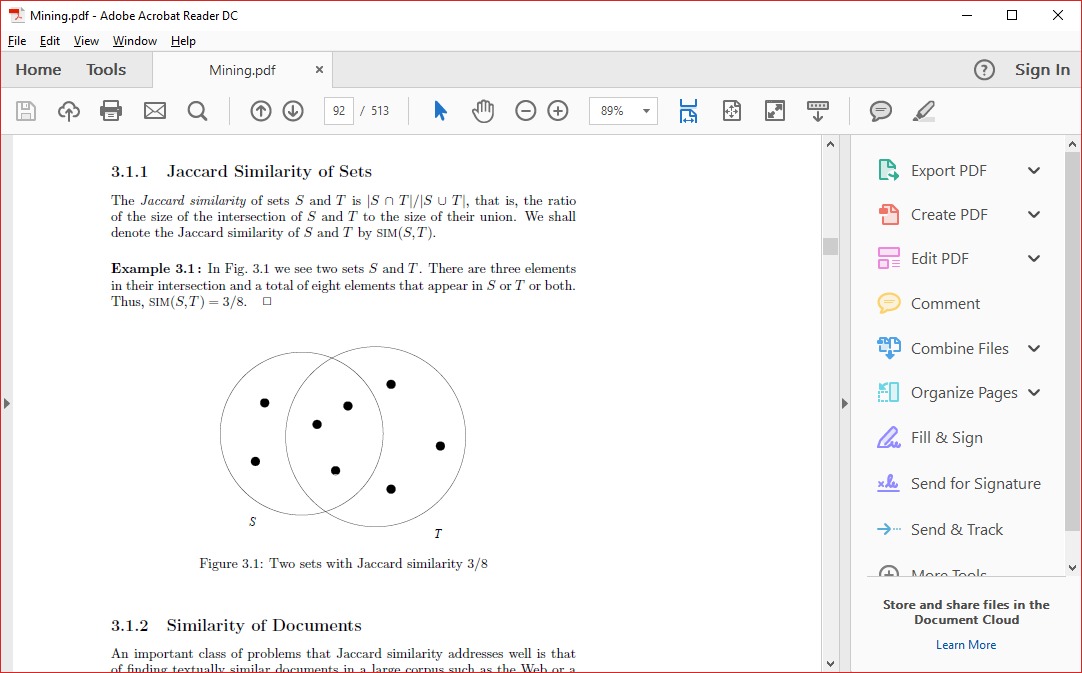
* The Jaccard similarity is a common index for binary variables.
* It is defined as the quotient between the intersection and the union of the pairwise compared variables among two objects.{\displaystyle d\_{J}(A,B)=1-J(A,B)={{|A\cup B|-|A\cap B|} \over |A\cup B|}.}

**Usage**

* The Jaccard similarity can be used, when interested in binary differences between two or more objects.
* Especially in ecological research investigations often focus on the presence/absence between several sites.
* When interested in characterising compared sites by the possibility of species to settle there abundances are often negligible.







**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.
* The aspect of similarity here is character-level similarity, not “similar meaning,” which requires us to examine the words in the documents and their uses.
* Textual similarity also has important uses. Many of these involve finding duplicates or near duplicates. First, let us observe that testing whether two documents are exact duplicates is easy; just compare the two documents character-by-character, and if they ever differ then they are not the same.
* However, in many applications, the documents are not identical, yet they share large portions of their text. Here are some examples:

**Plagiarism**

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

**Mirror Pages**

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

**Articles from the Same Source**

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

**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.

**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. 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, we might be able to discover that science fiction books are mutually similar and put them in one group.
* Then, we can get a more powerful notion of customer-similarity by asking whether they made purchases within many of the same groups.

**Movie Ratings**

* NetFlix records which movies each of its customers rented, and also the ratings assigned to those movies by the customers.
* We can see movies as similar if they were rented or rated highly by many of the same customers, and see customers as similar if they rented or rated highly many of the same movies.
* The same observations that we made for Amazon above apply in this situation: similarities need not be high to be significant, and clustering movies by genre will make things easier.
* When our data consists of ratings rather than binary decisions (bought/did not buy or liked/disliked), we cannot rely simply on sets as representations of customers or items. Some options are:
* Ignore low-rated customer/movie pairs; that is, treat these events as if the customer never watched the movie.
* When comparing customers, imagine two set elements for each movie, “liked” and “hated.” If a customer rated a movie highly, put the “liked” for that movie in the customer’s set.
* If they gave a low rating to a movie,put “hated” for that movie in their set. Then, we can look for high Jaccard similarity among these sets.
* We can do a similar trick when comparing movies.
* 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.

**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.

**k-Shingles**

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

**Example 1:** 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 2 :** 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.

**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 275 =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 20k.
* For large documents, such as research articles, choice k = 9 is considered safe.

**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 232 − 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.
* 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 232−1.
* 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 232 likely shingles. When we hash them down to four bytes, we can expect almost any sequence of four bytes to be possible

**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.
* 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.

**Matrix Representation of Sets**

Before explaining how it is possible to construct small signatures from large

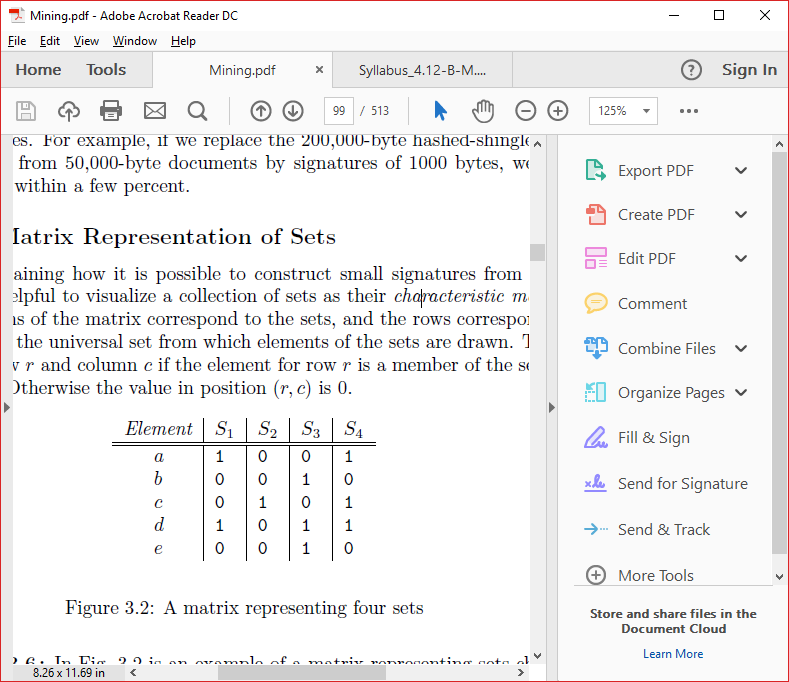
sets, it is helpful to visualize a collection of sets as their characteristic matrix.

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.



In Fig. 3.2 is an example of a matrix representing sets chosen from the universal set {a, b, c, d, e}. Here, S1 = {a, d}, S2 = {c}, S3 = {b, d, e},and S4 = {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.

**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.
* 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 :** Let us suppose we pick the order of rows beadc for the matrix

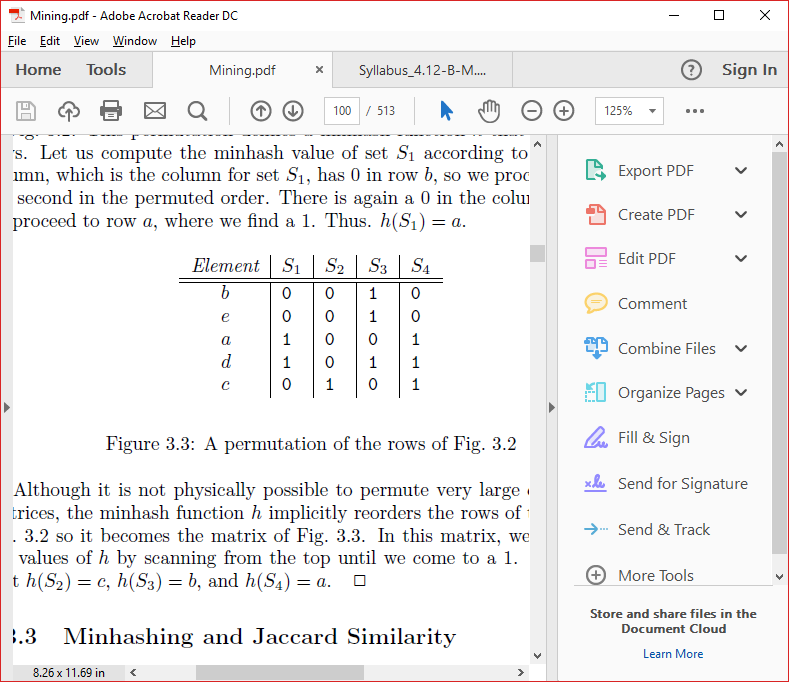
of Fig. 3.2. This permutation defines a minhash function h that maps sets to

rows. Let us compute the minhash value of set S1 according to h. The first

column, which is the column for set S1, 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 S1, so

we proceed to row a, where we find a 1. Thus. h(S1) = a.



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(S2) = c, h(S3) = b, and h(S4) = a.

**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 S1 and S2, then rows can be divided into three classes:

1. Type X rows have 1 in both columns.
2. Type Y rows have 1 in one of the columns and 0 in the other.
3. Type Z rows have 0 in both columns.

* Since the matrix is sparse, most rows are of type Z. However, it is the ratio of the numbers of type X and type Y rows that determine both SIM(S1, S2) and the probability that h(S1) = h(S2).
* Let there be x rows of type X and y rows of type Y . Then SIM(S1, S2) = x/(x + y). The reason is that x is the size of S1 ∩ S2 and x + y is the size of S1 ∪ S2.
* Now, consider the probability that h(S1) = h(S2). 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(S1) = h(S2).
* 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(S1) 6= h(S2) if we first meet a type Y row.
* We conclude the probability that h(S1) = h(S2) is x/(x + y), which is also the Jaccard similarity of S1 and S2.

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 h1, h2, . . . , hn.
* From the column representing set S, construct the minhash signature for S, the vector [h1(S), h2(S), . . . , hn(S)]. We normally represent this list of hash-values as a column. Thus, we can form from matrix M a signature matrix, in which the ith column of M is replaced by the minhash signature for (the set of) the ith column.
* Note that the signature matrix has the same number of columns as M but only n rows.
* Even if M is not represented explicitly, but in some compressed form suitable for a sparse matrix (e.g., by the locations of its 1’s), it is normal for the signature matrix to be much smaller than M.

**3.3.5 Computing Minhash Signatures**

* It is not feasible to permute a large characteristic matrix explicitly. Even picking a random permutation of millions or billions of rows is time-consuming, and the necessary sorting of the rows would take even more time. Thus, permuted matrices like that suggested by Fig. 3.3, while conceptually appealing, are not implementable.
* Fortunately, it is possible to simulate the effect of a random permutation by a random hash function that maps row numbers to as many buckets as there are rows. A hash function that maps integers 0, 1, . . . , 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 h1, h2, . . . , hn on the rows. We construct the signature matrix by considering each row in their given order.
* Let SIG(i, c) be the element of the signature matrix for the ith hash function and column c. Initially, set SIG(i, c) to ∞ for all i and c. We handle row r by doing the following:

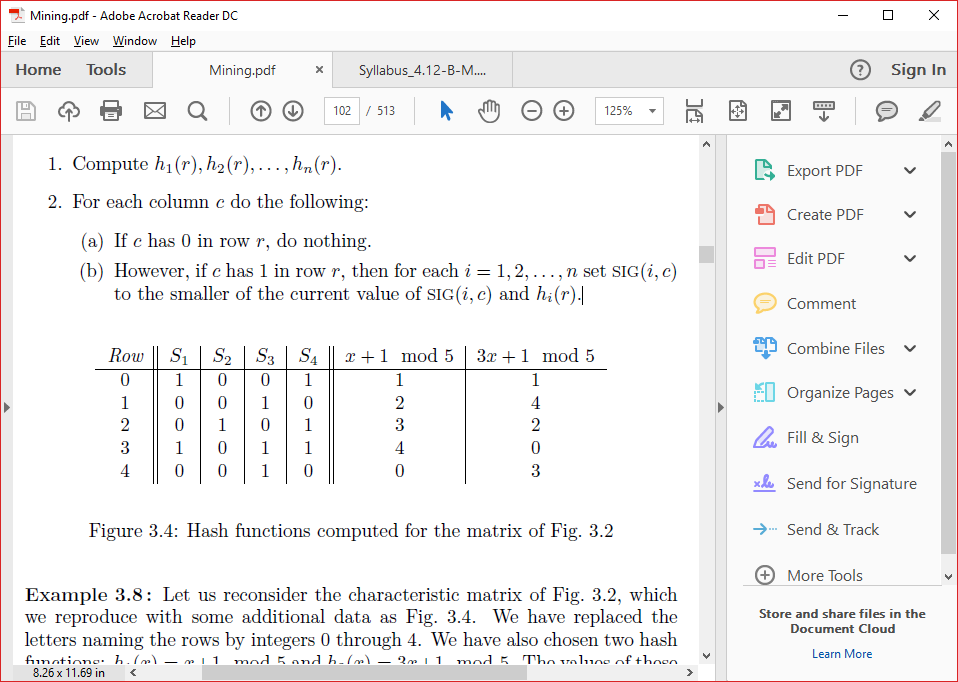
1. Compute h1(r), h2(r), . . . , hn(r).

2. For each column c do the following:

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

(b) However, if c has 1 in row r, then for each i = 1, 2, . . . , n set SIG(i, c)

to the smaller of the current value of SIG(i, c) and hi(r).



**3.4 Locality-Sensitive Hashing for Documents**

* Even though we can use minhashing to compress large documents into small signatures and preserve the expected similarity of any pair of documents, it still may be impossible to find the pairs with greatest similarity efficiently.
* The reason is that the number of pairs of documents may be too large, even if there are not too many documents.
* Example 3.9 : Suppose we have a million documents, and we use signatures of length 250. Then we use 1000 bytes per document for the signatures, and the entire data fits in a gigabyte – less than a typical main memory of a laptop.
* However, there are 1,000,0002 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.

**LSH for Minhash Signatures**

* One general approach to LSH is to “hash” items several times, in such a way that similar items are more likely to be hashed to the same bucket than dissimilar items are.
* We then consider any pair that hashed to the same bucket for any of the hashings to be a candidate pair.
* We check only the candidate pairs for similarity. The hope is that most of the dissimilar pairs will never hash to the same bucket, and therefore will never be checked.
* Those dissimilar pairs that do hash to the same bucket are false positives; we hope these will be only a small fraction of all pairs.
* We also hope that most of the truly similar pairs will hash to the same bucket under at least one of the hash functions.
* Those that do not are false negatives; we hope these will be only a small fraction of the truly similar pairs.
* If we have minhash signatures for the items, an effective way to choose the hashings is to divide the signature matrix into b bands consisting of r rows each.
* For each band, there is a hash function that takes vectors of r integers (the portion of one column within that band) and hashes them to some large number of buckets.
* We can use the same hash function for all the bands, but we use a separate bucket array for each band, so columns with the same vector in different bands will not hash to the same bucket.

**Example 3.10 :** Figure 3.6 shows part of a signature matrix of 12 rows divided

into four bands of three rows each. The second and fourth of the explicitly

shown columns each have the column vector [0, 2, 1] in the first band, so they

will definitely hash to the same bucket in the hashing for the first band. Thus,

regardless of what those columns look like in the other three bands, this pair

of columns will be a candidate pair. It is possible that other columns, such as

the first two shown explicitly, will also hash to the same bucket according to

the hashing of the first band. However, since their column vectors are different,

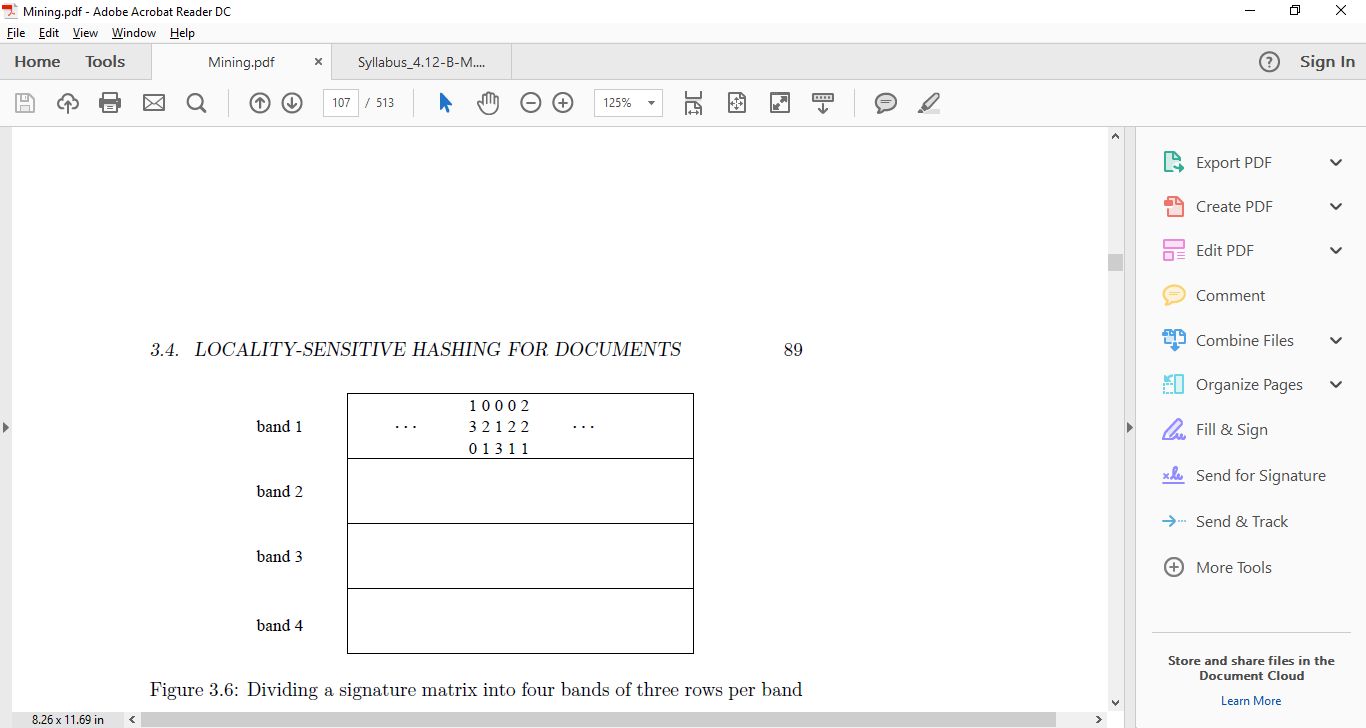
[1, 3, 0] and [0, 2, 1], and there are many buckets for each hashing, we expect the

chances of an accidental collision to be very small. We shall normally assume

that two vectors hash to the same bucket if and only if they are identical.

Two columns that do not agree in band 1 have three other chances to become

a candidate pair; they might be identical in any one of these other bands.



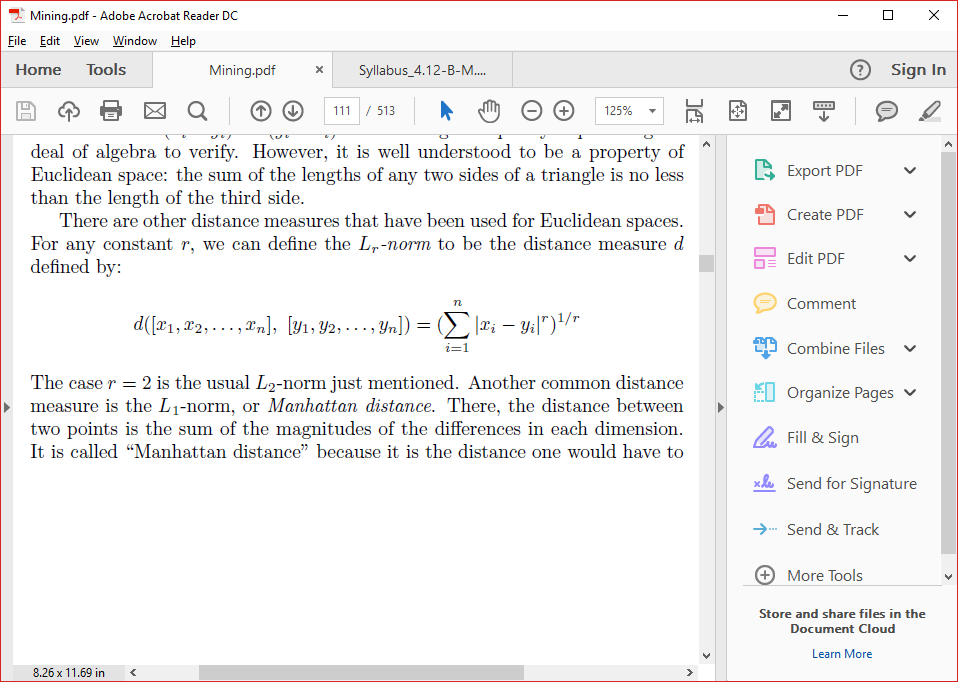
3.5.1 Definition of a Distance Measure

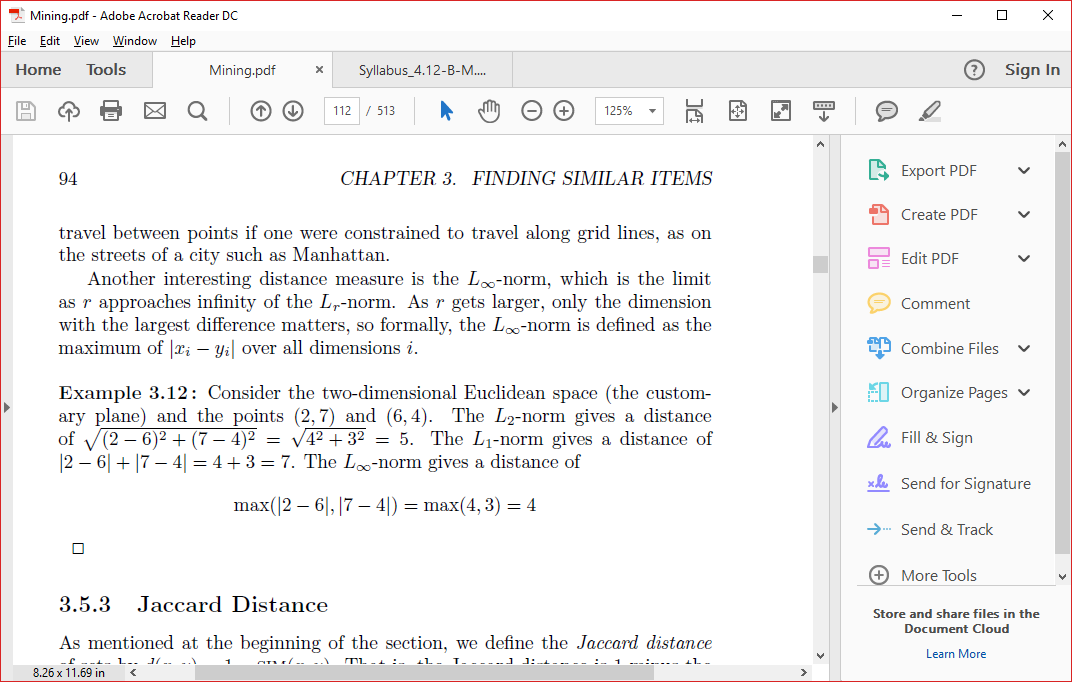
* Suppose we have a set of points, called a space. A distance measure on this space is a function d(x, y) that takes two points in the space as arguments and produces a real number, and satisfies the following axioms:

1. d(x, y) ≥ 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) ≤ 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 L2-norm, is defined:
  + - 
* The case r = 2 is the usual L2-norm just mentioned. Another common distance measure is the L1-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 Lr-norm. As r gets larger, only the dimension with the largest difference matters, so formally, the L∞-norm is defined as the maximum of |xi − yi| over all dimensions i.



**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 − SIM(x, y). That is, the Jaccard distance is 1 minus the

ratio of the sizes of the intersection and union of sets x and y. We must verify

that this function is a distance measure.

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

the size of the union.

2. d(x, y) = 0 if x = y, because x ∪ x = x ∩ x = x. However, if x 6= y, then

the size of x ∩ y is strictly less than the size of x ∪ y, so d(x, y) is strictly

positive.

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

x ∪ y = y ∪ x and x ∩ y = y ∩ x.

4. For the triangle inequality, 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) ≤ d(x, z) + d(z, y) to the statement that if

h is a random minhash function, then the probability that h(x) 6= h(y)

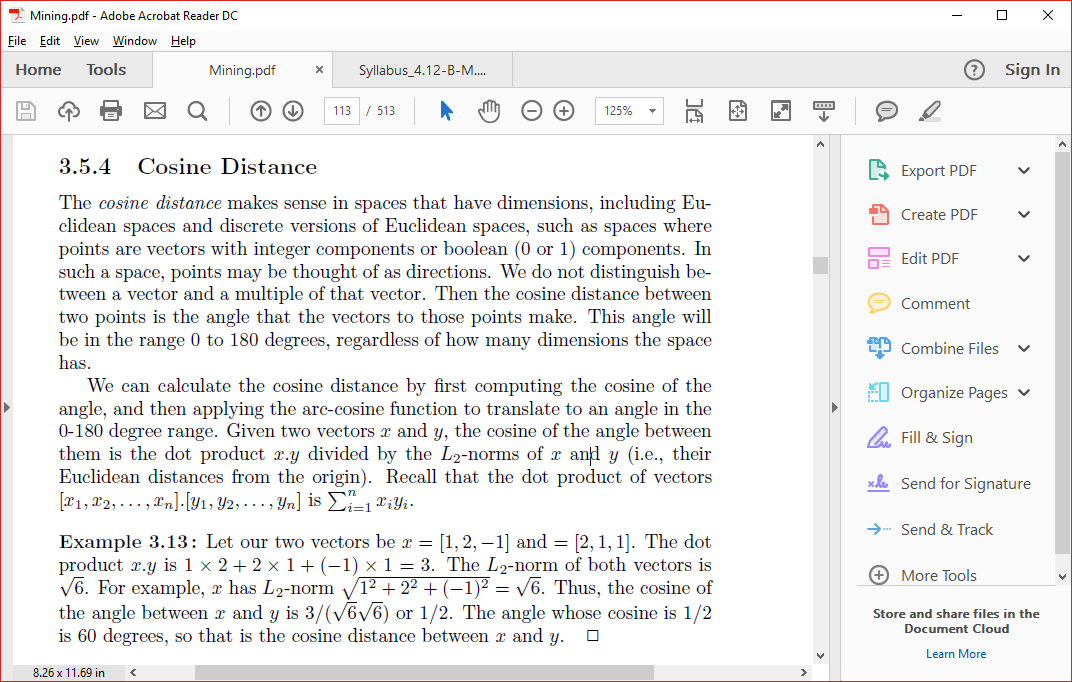
is no greater than the sum of the probability that h(x) 6= h(z) and the

probability that h(z) 6= h(y). However, this statement is true because

whenever h(x) 6= h(y), at least one of h(x) and h(y) must be different

from h(z). They could not both be h(z), because then h(x) and h(y)

would be the same.



**3.5.5 Edit Distance**

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

Example 3.14 : The edit distance between the strings x = abcde and y = acfdeg is 3. To convert x to y:

1. Delete b.
2. Insert f after c.
3. Insert g after e.
4. No sequence of fewer than three insertions and/or deletions will convert x to y.
5. Thus, d(x, y) = 3.

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

Example 3.15 : The strings x = abcde and y = acfdeg from Example 3.14

have a unique LCS, which is acde. We can be sure it is the longest possible,

because it contains every symbol appearing in both x and y. Fortunately, these

common symbols appear in the same order in both strings, so we are able to

use them all in an LCS. Note that the length of x is 5, the length of y is 6, and

the length of their LCS is 4. The edit distance is thus 5 + 6 − 2 × 4 = 3, which

agrees with the direct calculation in Example 3.14.

For another example, consider x = aba and y = bab. Their edit distance is

2. For example, we can convert x to y by deleting the first a and then inserting

b at the end. There are two LCS’s: ab and ba. Each can be obtained by

deleting one symbol from each string. As must be the case for multiple LCS’s

of the same pair of strings, both LCS’s have the same length. Therefore, we

may compute the edit distance as 3 + 3 − 2 × 2 = 2.

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

**3.8 Applications of Locality-Sensitive Hashing**

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

1**. Entity Resolution:** This term refers to matching data records that refer to

the same real-world entity, e.g., the same person. The principal problem

addressed here is that the similarity of records does not match exactly

either the similar-sets or similar-vectors models of similarity on which the

theory is built.

2. **Matching Fingerprints:** It is possible to represent fingerprints as sets.

However, we shall explore a different family of locality-sensitive hash functions

from the one we get by minhashing.

3. **Matching Newspaper Articles:** Here, we consider a different notion of

shingling that focuses attention on the core article in an on-line newspaper’s

Web page, ignoring all the extraneous material such as ads and

newspaper-specific material.

**3.8.1 Entity Resolution**

* It is common to have several data sets available, and to know that they refer to some of the same entities. For example, several different bibliographic sources provide information about many of the same books or papers. In the general case, we have records describing entities of some type, such as people or books.
* The records may all have the same format, or they may have different formats, with different kinds of information.
* There are many reasons why information about an entity may vary, even if the field in question is supposed to be the same. For example, names may be expressed differently in different records because of misspellings, absence of a middle initial, use of a nickname, and many other reasons. For example, “Bob S. Jomes” and “Robert Jones Jr.” may or may not be the same person.
* If records come from different sources, the fields may differ as well. One source’s records may have an “age” field, while another does not. The second source might have a “date of birth” field, or it may have no information at all about when a person was born.

**3.8.2 An Entity-Resolution Example**

* We shall examine a real example of how LSH was used to deal with an entity resolution problem. Company A was engaged by Company B to solicit customers for B. Company B would pay A a yearly fee, as long as the customer maintained their subscription. They later quarreled and disagreed over how many customers A had provided to B.
* Each had about 1,000,000 records, some of which described the same people; those were the customers A had provided to B. The records had different data fields, but unfortunately none of those fields was “this is a customer that A had provided to B.”
* Thus, the problem was to match records from the two sets to see if a pair represented the same person.
* Each record had fields for the name, address, and phone number of the person. However, the values in these fields could differ for many reasons. Not only were there the misspellings and other naming differences mentioned in Section 3.8.1, but there were other opportunities to disagree as well.
* A customer might give their home phone to A and their cell phone to B. Or they might move, and tell B but not A (because they no longer had need for a relationship with A). Area codes of phones sometimes change.
* The strategy for identifying records involved scoring the differences in three fields: name, address, and phone. To create a score describing the likelihood that two records, one from A and the other from B, described the same person, 100 points was assigned to each of the three fields, so records with exact matches in all three fields got a score of 300. However, there were deductions for mismatches in each of the three fields.
* As a first approximation, edit-distance (Section 3.5.5) was used, but the penalty grew quadratically with the distance.
* Then, certain publicly available tables were used to reduce the penalty in appropriate situations. For example, “Bill” and “William” were treated as if they differed in only one letter, even though their edit-distance is 5.
* However, it is not feasible to score all one trillion pairs of records. Thus, a simple LSH was used to focus on likely candidates. Three “hash functions” were used.
* The first sent records to the same bucket only if they had identical names; the second did the same but for identical addresses, and the third did the same for phone numbers. In practice, there was no hashing; rather the records were sorted by name, so records with identical names would appear consecutively and get scored for overall similarity of the name, address, and phone.
* Then the records were sorted by address, and those with the same 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

3.8.4 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.
* 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.

1. 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.

**3.8.6 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.
* We wish to ignore parts of the document, such as ads or the headlines of other articles to which the newspaper added a link, that are not part of the news article. It turns out that there is a noticeable difference between text that appears in prose and text that appears in ads or headlines. Prose has a much greater frequency of stop words, the very frequent words such as “the” or “and.”
* The total number of words that are considered stop words varies with the application, but it is common to use a list of several hundred of the most frequent words.
* **Example 3.23 :** A typical ad might say simply “Buy Sudzo.” On the other hand, a prose version of the same thought that might appear in an article is “I recommend that you buy Sudzo for your laundry.” In the latter sentence, it would be normal to treat “I,” “that,” “you,” “for,” and “your” as stop words.
* Suppose we define a shingle to be a stop word followed by the next two words. Then the ad “Buy Sudzo” from Example 3.23 has no shingles and would not be reflected in the representation of the Web page containing that ad. On the other hand, the sentence from Example 3.23 would be represented by five shingles: “I recommend that,” “that you buy,” “you buy Sudzo,” “for your laundry,” and “your laundry x,” where x is whatever word follows that sentence.
* Suppose we have two Web pages, each of which consists of half news text and half ads or other material that has a low density of stop words. If the news text is the same but the surrounding material is different, then we would expect that a large fraction of the shingles of the two pages would be the same.
* They might have a Jaccard similarity of 75%. However, if the surrounding material is the same but the news content is different, then the number of common shingles would be small, perhaps 25%. If we were to use the conventional shingling, where shingles are (say) sequences of 10 consecutive characters, we would expect the two documents to share half their shingles (i.e., a Jaccard similarity of 1/3), regardless of whether it as the news or the surrounding material that they shared.

**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.

**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 3.24 :** 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.

**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 Lx.
* Note that Ls ≤ Lt. The intersection of the sets represented by s and t cannot have more than Ls members, while their union has at least Lt members. Thus, the Jaccard similarity of s and t, which we denote SIM(s, t), is at most Ls/Lt.
* That is, in order for s and t to require comparison, it must be that J ≤ Ls/Lt, or equivalently, Lt ≤ Ls/J.

**Example 3.25 :** 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**

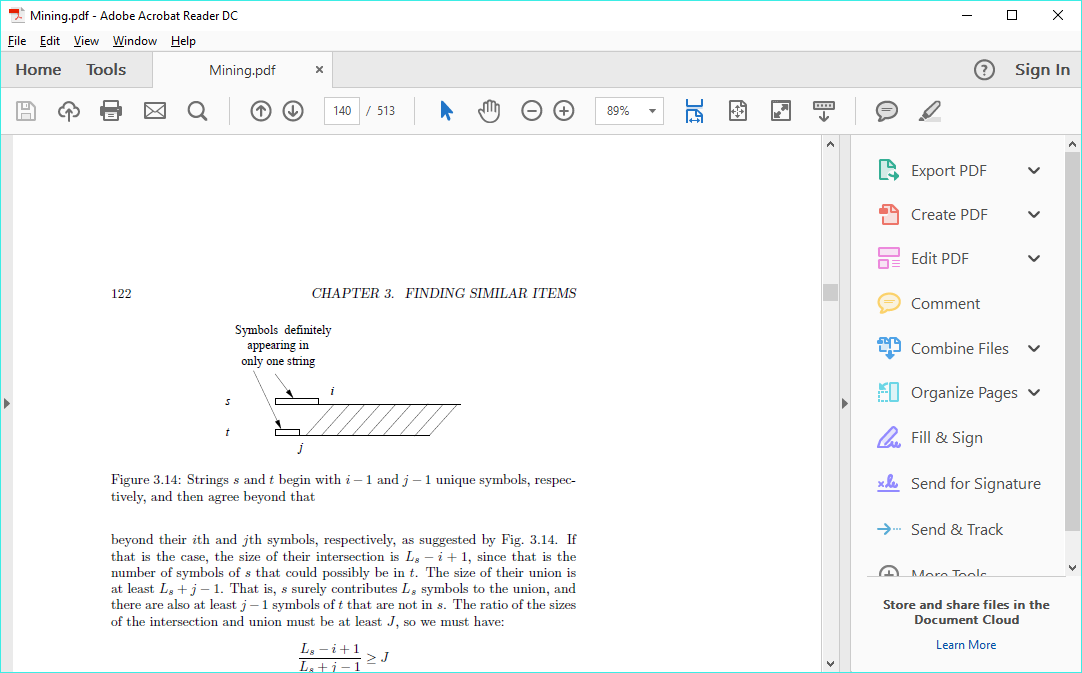
* In addition to length, there are several other features of strings that can be exploited to limit the number of comparisons that must be made to identify all pairs of similar strings.
* The simplest of these options is to create an index for each symbol; recall a symbol of a string is any one of the 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 Ls 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 SIM(s, t) ≥ J will have at least one symbol in its prefix that also appears in the prefix of s.
* Suppose not; rather SIM(s, t) ≥ 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 (Ls − p)/Ls.
* To be sure that we do not have to compare s with t, we must be certain that J > (Ls − p)/Ls. That is, p must be at least ⌊(1 − J)Ls⌋ + 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 = ⌊(1 − J)Ls⌋ + 1 to be the length of the prefix that gets indexed.

**Example 3.26** : Suppose J = 0.9. If Ls = 9, then p = ⌊0.1 × 9⌋ + 1 = ⌊0.9⌋ + 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. If t begins with a, and SIM(s, t) ≥ 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 Lt = 10, so t will be indexed under the symbols of its prefix of length ⌊0.1 × 10⌋ + 1 = 2.

* If t begins with c or a later letter, then the maximum value of SIM(s, t)
* occurs when t is cdefghij. But then 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; 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, ⌊(1 − J)Ls⌋ + 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 ⌊(1 − J)Ls⌋ + 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.
* 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. 3.14. If that is the case, the size of their intersection is Ls − 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 Ls + j − 1. That is, s surely contributes Ls 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:

Ls − i + 1

Ls + j − 1 ≥ J

* If we isolate j in this inequality, we have j ≤ 􀀀Ls(1 − J) − i + 1 + J\_/J.
* Example 3.27 : 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 ≤ (10 × 0.1 − 1 + 1 + 0.9)/0.9. That is, we only have to compare the symbol a with strings in the bucket for (a, j) if j ≤ 2.11. Thus, j can be 1 or 2, but nothing higher.
* Now suppose i = 2. Then we require j ≤ (10 × 0.1 − 2 + 1 + 0.9)/0.9, Or j ≤ 1. We conclude that we must look in the buckets for (a, 1), (a, 2), and (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 (a, j) and (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. 3.14, 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 3.28 :** The string s = acdefghijk, with J = 0.9, would be indexed in the buckets for (a, 1, 9) and (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 3.14 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 ≥ q. Here, the maximum size of the intersection is Ls − i + 1 − (p − q) Since Ls = i + p, we can write the above expression for the intersection size as q + 1.
* The minimum size of the union is Ls + j − 1, as it was when we did not take suffix length into account. Thus, we require q + 1 Ls + j − 1 ≥ J
* whenever p ≥ q.
* **Case 2:** p < q. Here, the maximum size of the intersection is Ls − i + 1, as when suffix length was not considered. However, the minimum size of the union is now Ls + j − 1 + q − p. If we again use the relationship Ls = i + p, we can replace Ls − 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 Ls − i + 1 i + j − 1 + q ≥ J whenever p < q. Example 3.29 : 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 Ls = 10. Since ⌊(1 − J)Ls⌋ + 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 ≥ q. The additional constraint we have on q and j is (q + 1)/(9 + j) ≥ 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 ≤ 9. Let us consider the possible values of q:

q = 9: We must have 10/(9 + j) ≥ 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) ≥ 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) ≥ 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 ≤ 8. Since the constraint (q+1)/(9+j) ≥ 0.8

does not depend on i,7 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 ≤ 7 and (q +1)/(9+j) ≥ 0.8. The

only option is q = 7 and j = 1.

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

11 − i

i + j + q − 1 ≥ 0.8

Again, consider each possible value of i.

i = 1: Then p = 9, so we require q ≥ 10 and 10/(q + j) ≥ 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 ≥ 9 and 9/(q + j + 1) ≥ 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 ≥ 8 and 8/(q + j + 2) ≥ 0.8. There are no

solutions.

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 3.15

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.