Chapter 1

Data Mining

In this intoductory chapter we begin with the essence of data mining and a discussion of how data mining is treated by the various disciplines that contribute to this field. We cover "Bonferroni's Principle," which is really a warning about overusing the ability to mine data. This chapter is also the place where we summarize a few useful ideas that are not data mining but are useful in understanding some important data-mining concepts. These include the TF.IDF measure of word importance, behavior of hash functions and indexes, and identities involving e, the base of natural logarithms. Finally, we give an outline of the topics covered in the balance of the book.

1.1 What is Data Mining?

The most commonly accepted definition of "data mining" is the discovery of "models" for data. A "model," however, can be one of several things. We mention below the most important directions in modeling.

1.1.1 Statistical Modeling

Statisticians were the first to use the term "data mining." Originally, "data mining" or "data dredging" was a derogatory term referring to attempts to extract information that was not supported by the data. Section 1.2 illustrates the sort of errors one can make by trying to extract what really isn't in the data. Today, "data mining" has taken on a positive meaning. Now, statisticians view data mining as the construction of a *statistical model*, that is, an underlying distribution from which the visible data is drawn.

Example 1.1 : Suppose our data is a set of numbers. This data is much simpler than data that would be data-mined, but it will serve as an example. A statistician might decide that the data comes from a Gaussian distribution and use a formula to compute the most likely parameters of this Gaussian. The mean

and standard deviation of this Gaussian distribution completely characterize the distribution and would become the model of the data. □

1.1.2 Machine Learning

There are some who regard data mining as synonymous with machine learning. There is no question that some data mining appropriately uses algorithms from machine learning. Machine-learning practitioners use the data as a training set, to train an algorithm of one of the many types used by machine-learning practitioners, such as Bayes nets, support-vector machines, decision trees, hidden Markov models, and many others.

There are situations where using data in this way makes sense. The typical case where machine learning is a good approach is when we have little idea of what we are looking for in the data. For example, it is rather unclear what it is about movies that makes certain movie-goers like or dislike it. Thus, in answering the "Netflix challenge" to devise an algorithm that predicts the ratings of movies by users, based on a sample of their responses, machine-learning algorithms have proved quite successful. We shall discuss a simple form of this type of algorithm in Section 9.4.

On the other hand, machine learning has not proved successful in situations where we can describe the goals of the mining more directly. An interesting case in point is the attempt by WhizBang! Labs[1] to use machine learning to locate people's resumes on the Web. It was not able to do better than algorithms designed by hand to look for some of the obvious words and phrases that appear in the typical resume. Since everyone who has looked at or written a resume has a pretty good idea of what resumes contain, there was no mystery about what makes a Web page a resume. Thus, there was no advantage to machine-learning over the direct design of an algorithm to discover resumes.

1.1.3 Computational Approaches to Modeling

More recently, computer scientists have looked at data mining as an algorithmic problem. In this case, the model of the data is simply the answer to a complex query about it. For instance, given the set of numbers of Example 1.1, we might compute their average and standard deviation. Note that these values might not be the parameters of the Gaussian that best fits the data, although they will almost certainly be very close if the size of the data is large.

There are many different approaches to modeling data. We have already mentioned the possibility of constructing a statistical process whereby the data could have been generated. Most other approaches to modeling can be described as either

1. Summarizing the data succinctly and approximately, or

[1]This startup attempted to use machine learning to mine large-scale data, and hired many of the top machine-learning people to do so. Unfortunately, it was not able to survive.

2. Extracting the most prominent features of the data and ignoring the rest.

We shall explore these two approaches in the following sections.

1.1.4 Summarization

One of the most interesting forms of summarization is the PageRank idea, which made Google successful and which we shall cover in Chapter 5. In this form of Web mining, the entire complex structure of the Web is summarized by a single number for each page. This number, the "PageRank" of the page, is (oversimplifying somewhat) the probability that a random walker on the graph would be at that page at any given time. The remarkable property this ranking has is that it reflects very well the "importance" of the page – the degree to which typical searchers would like that page returned as an answer to their search query.

Another important form of summary – clustering – will be covered in Chapter 7. Here, data is viewed as points in a multidimensional space. Points that are "close" in this space are assigned to the same cluster. The clusters themselves are summarized, perhaps by giving the centroid of the cluster and the average distance from the centroid of points in the cluster. These cluster summaries become the summary of the entire data set.

Example 1.2: A famous instance of clustering to solve a problem took place long ago in London, and it was done entirely without computers.[2] The physician John Snow, dealing with a Cholera outbreak plotted the cases on a map of the city. A small illustration suggesting the process is shown in Fig. 1.1.

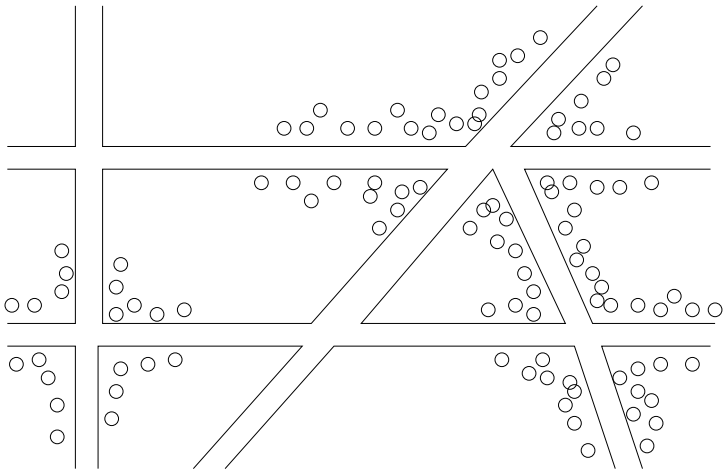

Figure 1.1: Plotting cholera cases on a map of London

[2]See `http://en.wikipedia.org/wiki/1854_Broad_Street_cholera_outbreak`.

The cases clustered around some of the intersections of roads. These intersections were the locations of wells that had become contaminated; people who lived nearest these wells got sick, while people who lived nearer to wells that had not been contaminated did not get sick. Without the ability to cluster the data, the cause of Cholera would not have been discovered. □

1.1.5 Feature Extraction

The typical feature-based model looks for the most extreme examples of a phenomenon and represents the data by these examples. If you are familiar with Bayes nets, a branch of machine learning and a topic we do not cover in this book, you know how a complex relationship between objects is represented by finding the strongest statistical dependencies among these objects and using only those in representing all statistical connections. Some of the important kinds of feature extraction from large-scale data that we shall study are:

1. *Frequent Itemsets.* This model makes sense for data that consists of "baskets" of small sets of items, as in the market-basket problem that we shall discuss in Chapter 6. We look for small sets of items that appear together in many baskets, and these "frequent itemsets" are the characterization of the data that we seek. The original application of this sort of mining was true market baskets: the sets of items, such as hamburger and ketchup, that people tend to buy together when checking out at the cash register of a store or super market.

2. *Similar Items.* Often, your data looks like a collection of sets, and the objective is to find pairs of sets that have a relatively large fraction of their elements in common. An example is treating customers at an online store like Amazon as the set of items they have bought. In order for Amazon to recommend something else they might like, Amazon can look for "similar" customers and recommend something many of these customers have bought. This process is called "collaborative filtering." If customers were single-minded, that is, they bought only one kind of thing, then clustering customers might work. However, since customers tend to have interests in many different things, it is more useful to find, for each customer, a small number of other customers who are similar in their tastes, and represent the data by these connections. We discuss similarity in Chapter 3.

1.2 Statistical Limits on Data Mining

A common sort of data-mining problem involves discovering unusual events hidden within massive amounts of data. This section is a discussion of the problem, including "Bonferroni's Principle," a warning against overzealous use of data mining.

1.2.1 Total Information Awareness

Following the terrorist attack of Sept. 11, 2001, it was noticed that there were four people enrolled in different flight schools, learning how to pilot commercial aircraft, although they were not affiliated with any airline. It was conjectured that the information needed to predict and foil the attack was available in data, but that there was then no way to examine the data and detect suspicious events. The response was a program called TIA, or *Total Information Awareness*, which was intended to mine all the data it could find, including credit-card receipts, hotel records, travel data, and many other kinds of information in order to track terrorist activity. TIA naturally caused great concern among privacy advocates, and the project was eventually killed by Congress. It is not the purpose of this book to discuss the difficult issue of the privacy-security tradeoff. However, the prospect of TIA or a system like it does raise many technical questions about its feasibility.

The concern raised by many is that if you look at so much data, and you try to find within it activities that look like terrorist behavior, are you not going to find many innocent activities – or even illicit activities that are not terrorism – that will result in visits from the police and maybe worse than just a visit? The answer is that it all depends on how narrowly you define the activities that you look for. Statisticians have seen this problem in many guises and have a theory, which we introduce in the next section.

1.2.2 Bonferroni's Principle

Suppose you have a certain amount of data, and you look for events of a certain type within that data. You can expect events of this type to occur, even if the data is completely random, and the number of occurrences of these events will grow as the size of the data grows. These occurrences are "bogus," in the sense that they have no cause other than that random data will always have some number of unusual features that look significant but aren't. A theorem of statistics, known as the *Bonferroni correction* gives a statistically sound way to avoid most of these bogus positive responses to a search through the data. Without going into the statistical details, we offer an informal version, *Bonferroni's principle*, that helps us avoid treating random occurrences as if they were real. Calculate the expected number of occurrences of the events you are looking for, on the assumption that data is random. If this number is significantly larger than the number of real instances you hope to find, then you must expect almost anything you find to be bogus, i.e., a statistical artifact rather than evidence of what you are looking for. This observation is the informal statement of Bonferroni's principle.

In a situation like searching for terrorists, where we expect that there are few terrorists operating at any one time, Bonferroni's principle says that we may only detect terrorists by looking for events that are so rare that they are unlikely to occur in random data. We shall give an extended example in the

next section.

1.2.3 An Example of Bonferroni's Principle

Suppose there are believed to be some "evil-doers" out there, and we want to detect them. Suppose further that we have reason to believe that periodically, evil-doers gather at a hotel to plot their evil. Let us make the following assumptions about the size of the problem:

1. There are one billion people who might be evil-doers.
2. Everyone goes to a hotel one day in 100.
3. A hotel holds 100 people. Hence, there are 100,000 hotels – enough to hold the 1% of a billion people who visit a hotel on any given day.
4. We shall examine hotel records for 1000 days.

To find evil-doers in this data, we shall look for people who, on two different days, were both at the same hotel. Suppose, however, that there really are no evil-doers. That is, everyone behaves at random, deciding with probability 0.01 to visit a hotel on any given day, and if so, choosing one of the 10^5 hotels at random. Would we find any pairs of people who appear to be evil-doers?

We can do a simple approximate calculation as follows. The probability of any two people both deciding to visit a hotel on any given day is .0001. The chance that they will visit the same hotel is this probability divided by 10^5, the number of hotels. Thus, the chance that they will visit the same hotel on one given day is 10^{-9}. The chance that they will visit the same hotel on two different given days is the square of this number, 10^{-18}. Note that the hotels can be different on the two days.

Now, we must consider how many events will indicate evil-doing. An "event" in this sense is a pair of people and a pair of days, such that the two people were at the same hotel on each of the two days. To simplify the arithmetic, note that for large n, $\binom{n}{2}$ is about $n^2/2$. We shall use this approximation in what follows. Thus, the number of pairs of people is $\binom{10^9}{2} = 5 \times 10^{17}$. The number of pairs of days is $\binom{1000}{2} = 5 \times 10^5$. The expected number of events that look like evil-doing is the product of the number of pairs of people, the number of pairs of days, and the probability that any one pair of people and pair of days is an instance of the behavior we are looking for. That number is

$$5 \times 10^{17} \times 5 \times 10^5 \times 10^{-18} = 250,000$$

That is, there will be a quarter of a million pairs of people who look like evil-doers, even though they are not.

Now, suppose there really are 10 pairs of evil-doers out there. The police will need to investigate a quarter of a million other pairs in order to find the real evil-doers. In addition to the intrusion on the lives of half a million innocent

people, the work involved is sufficiently great that this approach to finding evil-doers is probably not feasible.

1.2.4 Exercises for Section 1.2

Exercise 1.2.1: Using the information from Section 1.2.3, what would be the number of suspected pairs if the following changes were made to the data (and all other numbers remained as they were in that section)?

(a) The number of days of observation was raised to 2000.

(b) The number of people observed was raised to 2 billion (and there were therefore 200,000 hotels).

(c) We only reported a pair as suspect if they were at the same hotel at the same time on three different days.

! **Exercise 1.2.2:** Suppose we have information about the supermarket purchases of 100 million people. Each person goes to the supermarket 100 times in a year and buys 10 of the 1000 items that the supermarket sells. We believe that a pair of terrorists will buy exactly the same set of 10 items (perhaps the ingredients for a bomb?) at some time during the year. If we search for pairs of people who have bought the same set of items, would we expect that any such people found were truly terrorists?[3]

1.3 Things Useful to Know

In this section, we offer brief introductions to subjects that you may or may not have seen in your study of other courses. Each will be useful in the study of data mining. They include:

1. The TF.IDF measure of word importance.
2. Hash functions and their use.
3. Secondary storage (disk) and its effect on running time of algorithms.
4. The base e of natural logarithms and identities involving that constant.
5. Power laws.

[3]That is, assume our hypothesis that terrorists will surely buy a set of 10 items in common at some time during the year. We don't want to address the matter of whether or not terrorists would necessarily do so.

1.3.1 Importance of Words in Documents

In several applications of data mining, we shall be faced with the problem of categorizing documents (sequences of words) by their topic. Typically, topics are identified by finding the special words that characterize documents about that topic. For instance, articles about baseball would tend to have many occurrences of words like "ball," "bat," "pitch,", "run," and so on. Once we have classified documents to determine they are about baseball, it is not hard to notice that words such as these appear unusually frequently. However, until we have made the classification, it is not possible to identify these words as characteristic.

Thus, classification often starts by looking at documents, and finding the significant words in those documents. Our first guess might be that the words appearing most frequently in a document are the most significant. However, that intuition is exactly opposite of the truth. The most frequent words will most surely be the common words such as "the" or "and," which help build ideas but do not carry any significance themselves. In fact, the several hundred most common words in English (called *stop words*) are often removed from documents before any attempt to classify them.

In fact, the indicators of the topic are relatively rare words. However, not all rare words are equally useful as indicators. There are certain words, for example "notwithstanding" or "albeit," that appear rarely in a collection of documents, yet do not tell us anything useful. On the other hand, a word like "chukker" is probably equally rare, but tips us off that the document is about the sport of polo. The difference between rare words that tell us something and those that do not has to do with the concentration of the useful words in just a few documents. That is, the presence of a word like "albeit" in a document does not make it terribly more likely that it will appear multiple times. However, if an article mentions "chukker" once, it is likely to tell us what happened in the "first chukker," then the "second chukker," and so on. That is, the word is likely to be repeated if it appears at all.

The formal measure of how concentrated into relatively few documents are the occurrences of a given word is called TF.IDF (*Term Frequency times Inverse Document Frequency*). It is normally computed as follows. Suppose we have a collection of N documents. Define f_{ij} to be the *frequency* (number of occurrences) of term (word) i in document j. Then, define the *term frequency* TF_{ij} to be:

$$TF_{ij} = \frac{f_{ij}}{\max_k f_{kj}}$$

That is, the term frequency of term i in document j is f_{ij} normalized by dividing it by the maximum number of occurrences of any term (perhaps excluding stop words) in the same document. Thus, the most frequent term in document j gets a TF of 1, and other terms get fractions as their term frequency for this document.

The IDF for a term is defined as follows. Suppose term i appears in n_i

of the N documents in the collection. Then $IDF_i = \log_2(N/n_i)$. The TF.IDF score for term i in document j is then defined to be $TF_{ij} \times IDF_i$. The terms with the highest TF.IDF score are often the terms that best characterize the topic of the document.

Example 1.3: Suppose our repository consists of $2^{20} = 1{,}048{,}576$ documents. Suppose word w appears in $2^{10} = 1024$ of these documents. Then $IDF_w = \log_2(2^{20}/2^{10}) = \log 2(2^{10}) = 10$. Consider a document j in which w appears 20 times, and that is the maximum number of times in which any word appears (perhaps after eliminating stop words). Then $TF_{wj} = 1$, and the TF.IDF score for w in document j is 10.

Suppose that in document k, word w appears once, while the maximum number of occurrences of any word in this document is 20. Then $TF_{wk} = 1/20$, and the TF.IDF score for w in document k is $1/2$. □

1.3.2 Hash Functions

The reader has probably heard of hash tables, and perhaps used them in Java classes or similar packages. The hash functions that make hash tables feasible are also essential components in a number of data-mining algorithms, where the hash table takes an unfamiliar form. We shall review the basics here.

First, a hash function h takes a *hash-key* value as an argument and produces a *bucket number* as a result. The bucket number is an integer, normally in the range 0 to $B - 1$, where B is the number of buckets. Hash-keys can be of any type. There is an intuitive property of hash functions that they "randomize" hash-keys. To be precise, if hash-keys are drawn randomly from a reasonable population of possible hash-keys, then h will send approximately equal numbers of hash-keys to each of the B buckets. It would be impossible to do so if, for example, the population of possible hash-keys were smaller than B. Such a population would not be "reasonable." However, there can be more subtle reasons why a hash function fails to achieve an approximately uniform distribution into buckets.

Example 1.4: Suppose hash-keys are positive integers. A common and simple hash function is to pick $h(x) = x \mod B$, that is, the remainder when x is divided by B. That choice works fine if our population of hash-keys is all positive integers. $1/B$th of the integers will be assigned to each of the buckets. However, suppose our population is the even integers, and $B = 10$. Then only buckets 0, 2, 4, 6, and 8 can be the value of $h(x)$, and the hash function is distinctly nonrandom in its behavior. On the other hand, if we picked $B = 11$, then we would find that 1/11th of the even integers get sent to each of the 11 buckets, so the hash function would work very well. □

The generalization of Example 1.4 is that when hash-keys are integers, chosing B so it has any common factor with all (or even most of) the possible hash-keys will result in nonrandom distribution into buckets. Thus, it is normally

preferred that we choose B to be a prime. That choice reduces the chance of nonrandom behavior, although we still have to consider the possibility that all hash-keys have B as a factor. Of course there are many other types of hash functions not based on modular arithmetic. We shall not try to summarize the options here, but some sources of information will be mentioned in the bibliographic notes.

What if hash-keys are not integers? In a sense, all data types have values that are composed of bits, and sequences of bits can always be interpreted as integers. However, there are some simple rules that enable us to convert common types to integers. For example, if hash-keys are strings, convert each character to its ASCII or Unicode equivalent, which can be interpreted as a small integer. Sum the integers before dividing by B. As long as B is smaller than the typical sum of character codes for the population of strings, the distribution into buckets will be relatively uniform. If B is larger, then we can partition the characters of a string into groups of several characters each. Treat the concatenation of the codes for the characters of a group as a single integer. Sum the integers associated with all the groups of a string, and divide by B as before. For instance, if B is around a billion, or 2^{30}, then grouping characters four at a time will give us 32-bit integers. The sum of several of these will distribute fairly evenly into a billion buckets.

For more complex data types, we can extend the idea used for converting strings to integers, recursively.

- For a type that is a record, each of whose components has its own type, recursively convert the value of each component to an integer, using the algorithm appropriate for the type of that component. Sum the integers for the components, and convert the integer sum to buckets by dividing by B.
- For a type that is an array, set, or bag of elements of some one type, convert the values of the elements' type to integers, sum the integers, and divide by B.

1.3.3 Indexes

An *index* is a data structure that makes it efficient to retrieve objects given the value of one or more elements of those objects. The most common situation is one where the objects are records, and the index is on one of the fields of that record. Given a value v for that field, the index lets us retrieve all the records with value v in that field. For example, we could have a file of (name, address, phone) triples, and an index on the phone field. Given a phone number, the index allows us to find quickly the record or records with that phone number.

There are many ways to implement indexes, and we shall not attempt to survey the matter here. The bibliographic notes give suggestions for further reading. However, a hash table is one simple way to build an index. The field

or fields on which the index is based form the hash-key for a hash function. Records have the hash function applied to value of the hash-key, and the record itself is placed in the bucket whose number is determined by the hash function. The bucket could be a list of records in main-memory, or a disk block, for example.

Then, given a hash-key value, we can hash it, find the bucket, and need to search only that bucket to find the records with that value for the hash-key. If we choose the number of buckets B to be comparable to the number of records in the file, then there will be relatively few records in any bucket, and the search of a bucket takes little time.

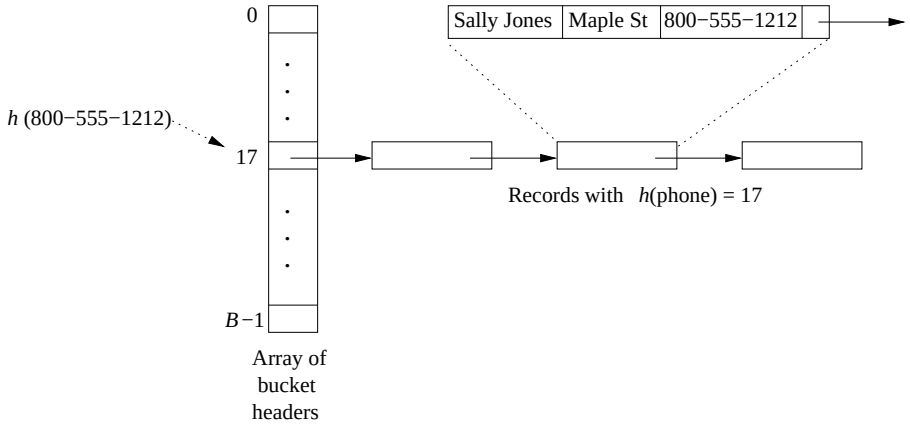

Figure 1.2: A hash table used as an index; phone numbers are hashed to buckets, and the entire record is placed in the bucket whose number is the hash value of the phone

Example 1.5 : Figure 1.2 suggests what a main-memory index of records with name, address, and phone fields might look like. Here, the index is on the phone field, and buckets are linked lists. We show the phone 800-555-1212 hashed to bucket number 17. There is an array of *bucket headers*, whose ith element is the head of a linked list for the bucket numbered i. We show expanded one of the elements of the linked list. It contains a record with name, address, and phone fields. This record is in fact one with the phone number 800-555-1212. Other records in that bucket may or may not have this phone number. We only know that whatever phone number they have is a phone that hashes to 17. □

1.3.4 Secondary Storage

It is important, when dealing with large-scale data, that we have a good understanding of the difference in time taken to perform computations when the data is initially on disk, as opposed to the time needed if the data is initially in

main memory. The physical characteristics of disks is another subject on which we could say much, but shall say only a little and leave the interested reader to follow the bibliographic notes.

Disks are organized into *blocks*, which are the minimum units that the operating system uses to move data between main memory and disk. For example, the Windows operating system uses blocks of 64K bytes (i.e., $2^{16} = 65{,}536$ bytes to be exact). It takes approximately ten milliseconds to *access* (move the disk head to the track of the block and wait for the block to rotate under the head) and read a disk block. That delay is at least five orders of magnitude (a factor of 10^5) slower than the time taken to read a word from main memory, so if all we want to do is access a few bytes, there is an overwhelming benefit to having data in main memory. In fact, if we want to do something simple to every byte of a disk block, e.g., treat the block as a bucket of a hash table and search for a particular value of the hash-key among all the records in that bucket, then the time taken to move the block from disk to main memory will be far larger than the time taken to do the computation.

By organizing our data so that related data is on a single *cylinder* (the collection of blocks reachable at a fixed radius from the center of the disk, and therefore accessible without moving the disk head), we can read all the blocks on the cylinder into main memory in considerably less than 10 milliseconds per block. You can assume that a disk cannot transfer data to main memory at more than a hundred million bytes per second, no matter how that data is organized. That is not a problem when your dataset is a megabyte. But a dataset of a hundred gigabytes or a terabyte presents problems just accessing it, let alone doing anything useful with it.

1.3.5 The Base of Natural Logarithms

The constant $e = 2.7182818\cdots$ has a number of useful special properties. In particular, e is the limit of $(1 + \frac{1}{x})^x$ as x goes to infinity. The values of this expression for $x = 1, 2, 3, 4$ are approximately $2, 2.25, 2.37, 2.44$, so you should find it easy to believe that the limit of this series is around 2.72.

Some algebra lets us obtain approximations to many seemingly complex expressions. Consider $(1+a)^b$, where a is small. We can rewrite the expression as $(1+a)^{(1/a)(ab)}$. Then substitute $a = 1/x$ and $1/a = x$, so we have $(1+\frac{1}{x})^{x(ab)}$, which is

$$\Big((1 + \frac{1}{x})^x\Big)^{ab}$$

Since a is assumed small, x is large, so the subexpression $(1 + \frac{1}{x})^x$ will be close to the limiting value of e. We can thus approximate $(1 + a)^b$ as e^{ab}.

Similar identities hold when a is negative. That is, the limit as x goes to infinity of $(1 - \frac{1}{x})^x$ is $1/e$. It follows that the approximation $(1 + a)^b = e^{ab}$ holds even when a is a small negative number. Put another way, $(1 - a)^b$ is approximately e^{-ab} when a is small and b is large.

Some other useful approximations follow from the Taylor expansion of e^x. That is, $e^x = \sum_{i=0}^{\infty} x^i/i!$, or $e^x = 1 + x + x^2/2 + x^3/6 + x^4/24 + \cdots$. When x is large, the above series converges slowly, although it does converge because $n!$ grows faster than x^n for any constant x. However, when x is small, either positive or negative, the series converges rapidly, and only a few terms are necessary to get a good approximation.

Example 1.6: Let $x = 1/2$. Then

$$e^{1/2} = 1 + \frac{1}{2} + \frac{1}{8} + \frac{1}{48} + \frac{1}{384} + \cdots$$

or approximately $e^{1/2} = 1.64844$.

Let $x = -1$. Then

$$e^{-1} = 1 - 1 + \frac{1}{2} - \frac{1}{6} + \frac{1}{24} - \frac{1}{120} + \frac{1}{720} - \frac{1}{5040} + \cdots$$

or approximately $e^{-1} = 0.36786$. □

1.3.6 Power Laws

There are many phenomena that relate two variables by a *power law*, that is, a linear relationship between the logarithms of the variables. Figure 1.3 suggests such a relationship. If x is the horizontal axis and y is the vertical axis, then the relationship is $\log_{10} y = 6 - 2\log_{10} x$.

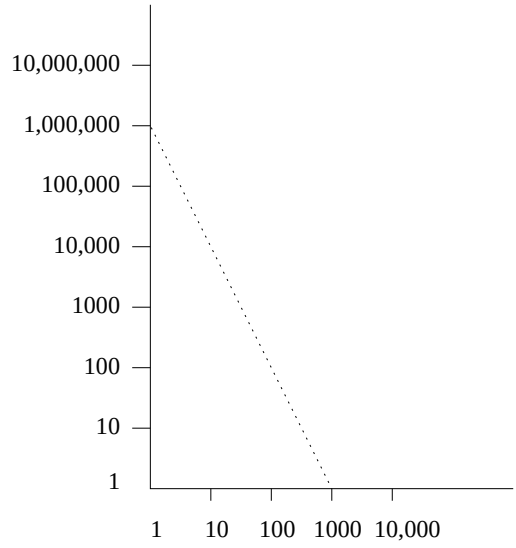

Figure 1.3: A power law with a slope of -2

The Matthew Effect

Often, the existence of power laws with values of the exponent higher than 1 are explained by the *Matthew effect*. In the biblical *Book of Matthew*, there is a verse about "the rich get richer." Many phenomena exhibit this behavior, where getting a high value of some property causes that very property to increase. For example, if a Web page has many links in, then people are more likely to find the page and may choose to link to it from one of their pages as well. As another example, if a book is selling well on Amazon, then it is likely to be advertised when customers go to the Amazon site. Some of these people will choose to buy the book as well, thus increasing the sales of this book.

Example 1.7: We might examine book sales at Amazon.com, and let x represent the rank of books by sales. Then y is the number of sales of the xth best-selling book over some period. The implication of the graph of Fig. 1.3 would be that the best-selling book sold 1,000,000 copies, the 10th best-selling book sold 10,000 copies, the 100th best-selling book sold 100 copies, and so on for all ranks between these numbers and beyond. The implication that above rank 1000 the sales are a fraction of a book is too extreme, and we would in fact expect the line to flatten out for ranks much higher than 1000. □

The general form of a power law relating x and y is $\log y = b + a \log x$. If we raise the base of the logarithm (which doesn't actually matter), say e, to the values on both sides of this equation, we get $y = e^b e^{a \log x} = e^b x^a$. Since e^b is just "some constant," let us replace it by constant c. Thus, a power law can be written as $y = cx^a$ for some constants a and c.

Example 1.8: In Fig. 1.3 we see that when $x = 1$, $y = 10^6$, and when $x = 1000$, $y = 1$. Making the first substitution, we see $10^6 = c$. The second substitution gives us $1 = c(1000)^a$. Since we now know $c = 10^6$, the second equation gives us $1 = 10^6(1000)^a$, from which we see $a = -2$. That is, the law expressed by Fig. 1.3 is $y = 10^6 x^{-2}$, or $y = 10^6/x^2$. □

We shall meet in this book many ways that power laws govern phenomena. Here are some examples:

1. *Node Degrees in the Web Graph*: Order all pages by the number of in-links to that page. Let x be the position of a page in this ordering, and let y be the number of in-links to the xth page. Then y as a function of x looks very much like Fig. 1.3. The exponent a is slightly larger than the -2 shown there; it has been found closer to 2.1.

2. *Sales of Products*: Order products, say books at Amazon.com, by their sales over the past year. Let y be the number of sales of the xth most popular book. Again, the function $y(x)$ will look something like Fig. 1.3. we shall discuss the consequences of this distribution of sales in Section 9.1.2, where we take up the matter of the "long tail."

3. *Sizes of Web Sites*: Count the number of pages at Web sites, and order sites by the number of their pages. Let y be the number of pages at the xth site. Again, the function $y(x)$ follows a power law.

4. *Zipf's Law*: This power law originally referred to the frequency of words in a collection of documents. If you order words by frequency, and let y be the number of times the xth word in the order appears, then you get a power law, although with a much shallower slope than that of Fig. 1.3. Zipf's observation was that $y = cx^{-1/2}$. Interestingly, a number of other kinds of data follow this particular power law. For example, if we order states in the US by population and let y be the population of the xth most populous state, then x and y obey Zipf's law approximately.

1.3.7 Exercises for Section 1.3

Exercise 1.3.1: Suppose there is a repository of ten million documents. What (to the nearest integer) is the IDF for a word that appears in (a) 40 documents (b) 10,000 documents?

Exercise 1.3.2: Suppose there is a repository of ten million documents, and word w appears in 320 of them. In a particular document d, the maximum number of occurrences of a word is 15. Approximately what is the TF.IDF score for w if that word appears (a) once (b) five times?

! **Exercise 1.3.3:** Suppose hash-keys are drawn from the population of all non-negative integers that are multiples of some constant c, and hash function $h(x)$ is $x \mod 15$. For what values of c will h be a suitable hash function, i.e., a large random choice of hash-keys will be divided roughly equally into buckets?

Exercise 1.3.4: In terms of e, give approximations to

(a) $(1.01)^{500}$ (b) $(1.05)^{1000}$ (c) $(0.9)^{40}$

Exercise 1.3.5: Use the Taylor expansion of e^x to compute, to three decimal places: (a) $e^{1/10}$ (b) $e^{-1/10}$ (c) e^2.

1.4 Outline of the Book

This section gives brief summaries of the remaining chapters of the book.

Chapter 2 is not about data mining per se. Rather, it introduces us to the MapReduce methodology for exploiting parallelism in computing clouds (racks

of interconnected processors). There is reason to believe that cloud computing, and MapReduce in particular, will become the normal way to compute when analysis of very large amounts of data is involved. A pervasive issue in later chapters will be the exploitation of the MapReduce methodology to implement the algorithms we cover.

Chapter 3 is about finding similar items. Our starting point is that items can be represented by sets of elements, and similar sets are those that have a large fraction of their elements in common. The key techniques of minhashing and locality-sensitive hashing are explained. These techniques have numerous applications and often give surprisingly efficient solutions to problems that appear impossible for massive data sets.

In Chapter 4, we consider data in the form of a stream. The difference between a stream and a database is that the data in a stream is lost if you do not do something about it immediately. Important examples of streams are the streams of search queries at a search engine or clicks at a popular Web site. In this chapter, we see several of the surprising applications of hashing that make management of stream data feasible.

Chapter 5 is devoted to a single application: the computation of PageRank. This computation is the idea that made Google stand out from other search engines, and it is still an essential part of how search engines know what pages the user is likely to want to see. Extensions of PageRank are also essential in the fight against spam (euphemistically called "search engine optimization"), and we shall examine the latest extensions of the idea for the purpose of combating spam.

Then, Chapter 6 introduces the market-basket model of data, and its canonical problems of association rules and finding frequent itemsets. In the market-basket model, data consists of a large collection of baskets, each of which contains a small set of items. We give a sequence of algorithms capable of finding all frequent pairs of items, that is pairs of items that appear together in many baskets. Another sequence of algorithms are useful for finding most of the frequent itemsets larger than pairs, with high efficiency.

Chapter 7 examines the problem of clustering. We assume a set of items with a distance measure defining how close or far one item is from another. The goal is to examine a large amount of data and partition it into subsets (clusters), each cluster consisting of items that are all close to one another, yet far from items in the other clusters.

Chapter 8 is devoted to on-line advertising and the computational problems it engenders. We introduce the notion of an on-line algorithm – one where a good response must be given immediately, rather than waiting until we have seen the entire dataset. The idea of competitive ratio is another important concept covered in this chapter; it is the ratio of the guaranteed performance of an on-line algorithm compared with the performance of the optimal algorithm that is allowed to see all the data before making any decisions. These ideas are used to give good algorithms that match bids by advertisers for the right to display their ad in response to a query against the search queries arriving at a

search engine.

Chapter 9 is devoted to recommendation systems. Many Web applications involve advising users on what they might like. The Netflix challenge is one example, where it is desired to predict what movies a user would like, or Amazon's problem of pitching a product to a customer based on information about what they might be interested in buying. There are two basic approaches to recommendation. We can characterize items by features, e.g., the stars of a movie, and recommend items with the same features as those the user is known to like. Or, we can look at other users with preferences similar to that of the user in question, and see what they liked (a technique known as collaborative filtering).

In Chapter 10, we study social networks and algorithms for their analysis. The canonical example of a social network is the graph of Facebook friends, where the nodes are people, and edges connect two people if they are friends. Directed graphs, such as followers on Twitter, can also be viewed as social networks. A common example of a problem to be addressed is identifying "communities," that is, small sets of nodes with an unusually large number of edges among them. Other questions about social networks are general questions about graphs, such as computing the transitive closure or diameter of a graph, but are made more difficult by the size of typical networks.

Chapter 11 looks at dimensionality reduction. We are given a very large matrix, typically sparse. Think of the matrix as representing a relationship between two kinds of entities, e.g., ratings of movies by viewers. Intuitively, there are a small number of concepts, many fewer concepts than there are movies or viewers, that explain why certain viewers like certain movies. We offer several algorithms that simplify matrices by decomposing them into a product of matrices that are much smaller in one of the two dimensions. One matrix relates entities of one kind to the small number of concepts and another relates the concepts to the other kind of entity. If done correctly, the product of the smaller matrices will be very close to the original matrix.

Finally, Chapter 12 discusses algorithms for machine learning from very large datasets. Techniques covered include perceptrons, support-vector machines, finding models by gradient descent, nearest-neighbor models, and decision trees.

1.5 Summary of Chapter 1

- ✦ *Data Mining*: This term refers to the process of extracting useful models of data. Sometimes, a model can be a summary of the data, or it can be the set of most extreme features of the data.

- ✦ *Bonferroni's Principle*: If we are willing to view as an interesting feature of data something of which many instances can be expected to exist in random data, then we cannot rely on such features being significant.

This observation limits our ability to mine data for features that are not sufficiently rare in practice.

- *TF.IDF*: The measure called TF.IDF lets us identify words in a collection of documents that are useful for determining the topic of each document. A word has high TF.IDF score in a document if it appears in relatively few documents, but appears in this one, and when it appears in a document it tends to appear many times.

- *Hash Functions*: A hash function maps hash-keys of some data type to integer bucket numbers. A good hash function distributes the possible hash-key values approximately evenly among buckets. Any data type can be the domain of a hash function.

- *Indexes*: An index is a data structure that allows us to store and retrieve data records efficiently, given the value in one or more of the fields of the record. Hashing is one way to build an index.

- *Storage on Disk*: When data must be stored on disk (secondary memory), it takes very much more time to access a desired data item than if the same data were stored in main memory. When data is large, it is important that algorithms strive to keep needed data in main memory.

- *Power Laws*: Many phenomena obey a law that can be expressed as $y = cx^a$ for some power a, often around -2. Such phenomena include the sales of the xth most popular book, or the number of in-links to the xth most popular page.

1.6 References for Chapter 1

[7] is a clear introduction to the basics of data mining. [2] covers data mining principally from the point of view of machine learning and statistics.

For construction of hash functions and hash tables, see [4]. Details of the TF.IDF measure and other matters regarding document processing can be found in [5]. See [3] for more on managing indexes, hash tables, and data on disk.

Power laws pertaining to the Web were explored by [1]. The Matthew effect was first observed in [6].

1. A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Weiner, "Graph structure in the web," *Computer Networks* **33**:1–6, pp. 309–320, 2000.

2. M.M. Gaber, *Scientific Data Mining and Knowledge Discovery — Principles and Foundations*, Springer, New York, 2010.

3. H. Garcia-Molina, J.D. Ullman, and J. Widom, *Database Systems: The Complete Book* Second Edition, Prentice-Hall, Upper Saddle River, NJ, 2009.

4. D.E. Knuth, *The Art of Computer Programming* Vol. 3 (*Sorting and Searching*), Second Edition, Addison-Wesley, Upper Saddle River, NJ, 1998.

5. C.P. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*, Cambridge Univ. Press, 2008.

6. R.K. Merton, "The Matthew effect in science," *Science* **159**:3810, pp. 56–63, Jan. 5, 1968.

7. P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining*, Addison-Wesley, Upper Saddle River, NJ, 2005.

Chapter 3

Finding Similar Items

A fundamental data-mining problem is to examine data for "similar" items. We shall take up applications in Section 3.1, but an example would be looking at a collection of Web pages and finding near-duplicate pages. These pages could be plagiarisms, for example, or they could be mirrors that have almost the same content but differ in information about the host and about other mirrors.

We begin by phrasing the problem of similarity as one of finding sets with a relatively large intersection. We show how the problem of finding textually similar documents can be turned into such a set problem by the technique known as "shingling." Then, we introduce a technique called "minhashing," which compresses large sets in such a way that we can still deduce the similarity of the underlying sets from their compressed versions. Other techniques that work when the required degree of similarity is very high are covered in Section 3.9.

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

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

3.1 Applications of Near-Neighbor Search

We shall focus initially on a particular notion of "similarity": the similarity of sets by looking at the relative size of their intersection. This notion of similarity is called "Jaccard similarity," and will be introduced in Section 3.1.1. We then examine some of the uses of finding similar sets. These include finding textually similar documents and collaborative filtering by finding similar customers and similar products. In order to turn the problem of textual similarity of documents

into one of set intersection, we use a technique called "shingling," which is introduced in Section 3.2.

3.1.1 Jaccard Similarity of Sets

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

Example 3.1 : In Fig. 3.1 we see two sets S and T. There are three elements in their intersection and a total of eight elements that appear in S or T or both. Thus, $\text{SIM}(S, T) = 3/8$. □

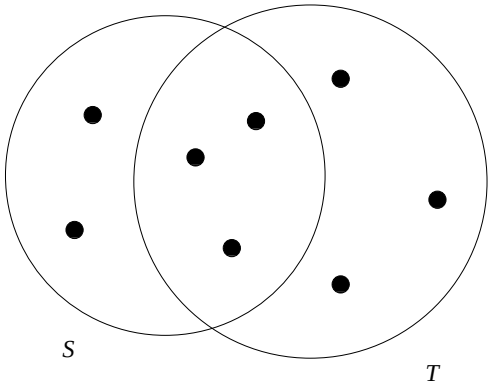

Figure 3.1: Two sets with Jaccard similarity 3/8

3.1.2 Similarity of Documents

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

Plagiarism

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

Mirror Pages

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

Articles from the Same Source

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

3.1.3 Collaborative Filtering as a Similar-Sets Problem

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

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

On-Line Purchases

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

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

Movie Ratings

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

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

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

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

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

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

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

3.1.4 Exercises for Section 3.1

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

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

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

3.2 Shingling of Documents

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

3.2.1 k-Shingles

A document is a string of characters. Define a k-shingle for a document to be any substring of length k found within the document. Then, we may associate

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

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

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

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

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

Example 3.4: If we use $k = 9$, but eliminate whitespace altogether, then we would see some lexical similarity in the sentences "`The plane was ready for touch down`". and "`The quarterback scored a touchdown`". However, if we retain the blanks, then the first has shingles `touch dow` and `ouch down`, while the second has `touchdown`. If we eliminated the blanks, then both would have `touchdown`. □

3.2.2 Choosing the Shingle Size

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

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

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

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

However, the calculation is a bit more subtle. Surely, more than 27 characters appear in emails, However, all characters do not appear with equal probability. Common letters and blanks dominate, while "z" and other letters that

have high point-value in Scrabble are rare. Thus, even short emails will have many 5-shingles consisting of common letters, and the chances of unrelated emails sharing these common shingles is greater than would be implied by the calculation in the paragraph above. A good rule of thumb is to imagine that there are only 20 characters and estimate the number of k-shingles as 20^k. For large documents, such as research articles, choice $k = 9$ is considered safe.

3.2.3 Hashing Shingles

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

Notice that we can differentiate documents better if we use 9-shingles and hash them down to four bytes than to use 4-shingles, even though the space used to represent a shingle is the same. The reason was touched upon in Section 3.2.2. If we use 4-shingles, most sequences of four bytes are unlikely or impossible to find in typical documents. Thus, the effective number of different shingles is much less than $2^{32} - 1$. If, as in Section 3.2.2, we assume only 20 characters are frequent in English text, then the number of different 4-shingles that are likely to occur is only $(20)^4 = 160{,}000$. However, if we use 9-shingles, there are many more than 2^{32} likely shingles. When we hash them down to four bytes, we can expect almost any sequence of four bytes to be possible, as was discussed in Section 1.3.2.

3.2.4 Shingles Built from Words

An alternative form of shingle has proved effective for the problem of identifying similar news articles, mentioned in Section 3.1.2. The exploitable distinction for this problem is that the news articles are written in a rather different style than are other elements that typically appear on the page with the article. News articles, and most prose, have a lot of stop words (see Section 1.3.1), the most common words such as "and," "you," "to," and so on. In many applications, we want to ignore stop words, since they don't tell us anything useful about the article, such as its topic.

However, for the problem of finding similar news articles, it was found that defining a shingle to be a stop word followed by the next two words, regardless of whether or not they were stop words, formed a useful set of shingles. The advantage of this approach is that the news article would then contribute more shingles to the set representing the Web page than would the surrounding ele-

ments. Recall that the goal of the exercise is to find pages that had the same articles, regardless of the surrounding elements. By biasing the set of shingles in favor of the article, pages with the same article and different surrounding material have higher Jaccard similarity than pages with the same surrounding material but with a different article.

Example 3.5: An ad might have the simple text "`Buy Sudzo`." However, a news article with the same idea might read something like "*A* `spokesperson` *for the* `Sudzo Corporation revealed today` *that* `studies` *have* `shown` *it is* `good` *for* `people` *to* `buy Sudzo products`." Here, we have italicized all the likely stop words, although there is no set number of the most frequent words that should be considered stop words. The first three shingles made from a stop word and the next two following are:

```
A spokesperson for
for the Sudzo
the Sudzo Corporation
```

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

3.2.5 Exercises for Section 3.2

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

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

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

3.3 Similarity-Preserving Summaries of Sets

Sets of shingles are large. Even if we hash them to four bytes each, the space needed to store a set is still roughly four times the space taken by the document. If we have millions of documents, it may well not be possible to store all the shingle-sets in main memory.[3]

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

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

the signatures give the exact similarity of the sets they represent, but the estimates they provide are close, and the larger the signatures the more accurate the estimates. For example, if we replace the 200,000-byte hashed-shingle sets that derive from 50,000-byte documents by signatures of 1000 bytes, we can usually get within a few percent.

3.3.1 Matrix Representation of Sets

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

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

Figure 3.2: A matrix representing four sets

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

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

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

3.3.2 Minhashing

The signatures we desire to construct for sets are composed of the results of a large number of calculations, say several hundred, each of which is a "minhash"

of the characteristic matrix. In this section, we shall learn how a minhash is computed in principle, and in later sections we shall see how a good approximation to the minhash is computed in practice.

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

Example 3.7: Let us suppose we pick the order of rows *beadc* for the matrix of Fig. 3.2. This permutation defines a minhash function h that maps sets to rows. Let us compute the minhash value of set S_1 according to h. The first column, which is the column for set S_1, has 0 in row b, so we proceed to row e, the second in the permuted order. There is again a 0 in the column for S_1, so we proceed to row a, where we find a 1. Thus. $h(S_1) = a$.

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

Figure 3.3: A permutation of the rows of Fig. 3.2

Although it is not physically possible to permute very large characteristic matrices, the minhash function h implicitly reorders the rows of the matrix of Fig. 3.2 so it becomes the matrix of Fig. 3.3. In this matrix, we can read off the values of h by scanning from the top until we come to a 1. Thus, we see that $h(S_2) = c$, $h(S_3) = b$, and $h(S_4) = a$. □

3.3.3 Minhashing and Jaccard Similarity

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

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

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

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

3. Type Z rows have 0 in both columns.

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

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

3.3.4 Minhash Signatures

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

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

3.3.5 Computing Minhash Signatures

It is not feasible to permute a large characteristic matrix explicitly. Even picking a random permutation of millions or billions of rows is time-consuming, and the necessary sorting of the rows would take even more time. Thus, permuted matrices like that suggested by Fig. 3.3, while conceptually appealing, are not implementable.

Fortunately, it is possible to simulate the effect of a random permutation by a random hash function that maps row numbers to as many buckets as there are rows. A hash function that maps integers $0, 1, \ldots, k-1$ to bucket numbers 0 through $k-1$ typically will map some pairs of integers to the same bucket and leave other buckets unfilled. However, the difference is unimportant as long as

k is large and there are not too many collisions. We can maintain the fiction that our hash function h "permutes" row r to position $h(r)$ in the permuted order.

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

1. Compute $h_1(r), h_2(r), \ldots, h_n(r)$.
2. For each column c do the following:
   (a) If c has 0 in row r, do nothing.
   (b) However, if c has 1 in row r, then for each $i = 1, 2, \ldots, n$ set $\text{SIG}(i, c)$ to the smaller of the current value of $\text{SIG}(i, c)$ and $h_i(r)$.

Row	S_1	S_2	S_3	S_4	$x + 1 \mod 5$	$3x + 1 \mod 5$
0	1	0	0	1	1	1
1	0	0	1	0	2	4
2	0	1	0	1	3	2
3	1	0	1	1	4	0
4	0	0	1	0	0	3

Figure 3.4: Hash functions computed for the matrix of Fig. 3.2

Example 3.8: Let us reconsider the characteristic matrix of Fig. 3.2, which we reproduce with some additional data as Fig. 3.4. We have replaced the letters naming the rows by integers 0 through 4. We have also chosen two hash functions: $h_1(x) = x+1 \mod 5$ and $h_2(x) = 3x+1 \mod 5$. The values of these two functions applied to the row numbers are given in the last two columns of Fig. 3.4. Notice that these simple hash functions are true permutations of the rows, but a true permutation is only possible because the number of rows, 5, is a prime. In general, there will be collisions, where two rows get the same hash value.

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

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

First, we consider row 0 of Fig. 3.4. We see that the values of $h_1(0)$ and $h_2(0)$ are both 1. The row numbered 0 has 1's in the columns for sets S_1 and

S_4, so only these columns of the signature matrix can change. As 1 is less than ∞, we do in fact change both values in the columns for S_1 and S_4. The current estimate of the signature matrix is thus:

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

Now, we move to the row numbered 1 in Fig. 3.4. This row has 1 only in S_3, and its hash values are $h_1(1) = 2$ and $h_2(1) = 4$. Thus, we set $\text{SIG}(1,3)$ to 2 and $\text{SIG}(2,3)$ to 4. All other signature entries remain as they are because their columns have 0 in the row numbered 1. The new signature matrix:

	S_1	S_2	S_3	S_4
h_1	1	∞	2	1
h_2	1	∞	4	1

The row of Fig. 3.4 numbered 2 has 1's in the columns for S_2 and S_4, and its hash values are $h_1(2) = 3$ and $h_2(2) = 2$. We could change the values in the signature for S_4, but the values in this column of the signature matrix, $[1, 1]$, are each less than the corresponding hash values $[3, 2]$. However, since the column for S_2 still has ∞'s, we replace it by $[3, 2]$, resulting in:

	S_1	S_2	S_3	S_4
h_1	1	3	2	1
h_2	1	2	4	1

Next comes the row numbered 3 in Fig. 3.4. Here, all columns but S_2 have 1, and the hash values are $h_1(3) = 4$ and $h_2(3) = 0$. The value 4 for h_1 exceeds what is already in the signature matrix for all the columns, so we shall not change any values in the first row of the signature matrix. However, the value 0 for h_2 is less than what is already present, so we lower $\text{SIG}(2,1)$, $\text{SIG}(2,3)$ and $\text{SIG}(2,4)$ to 0. Note that we cannot lower $\text{SIG}(2,2)$ because the column for S_2 in Fig. 3.4 has 0 in the row we are currently considering. The resulting signature matrix:

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

Finally, consider the row of Fig. 3.4 numbered 4. $h_1(4) = 0$ and $h_2(4) = 3$. Since row 4 has 1 only in the column for S_3, we only compare the current signature column for that set, $[2, 0]$ with the hash values $[0, 3]$. Since $0 < 2$, we change $\text{SIG}(1,3)$ to 0, but since $3 > 0$ we do not change $\text{SIG}(2,3)$. The final signature matrix is:

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

We can estimate the Jaccard similarities of the underlying sets from this signature matrix. Notice that columns 1 and 4 are identical, so we guess that $\text{SIM}(S_1, S_4) = 1.0$. If we look at Fig. 3.4, we see that the true Jaccard similarity of S_1 and S_4 is 2/3. Remember that the fraction of rows that agree in the signature matrix is only an estimate of the true Jaccard similarity, and this example is much too small for the law of large numbers to assure that the estimates are close. For additional examples, the signature columns for S_1 and S_3 agree in half the rows (true similarity 1/4), while the signatures of S_1 and S_2 estimate 0 as their Jaccard similarity (the correct value). □

3.3.6 Exercises for Section 3.3

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

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

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

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

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

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

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

Figure 3.5: Matrix for Exercise 3.3.3

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

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

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

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

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

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

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

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

3.4 Locality-Sensitive Hashing for Documents

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

Example 3.9: Suppose we have a million documents, and we use signatures of length 250. Then we use 1000 bytes per document for the signatures, and the entire data fits in a gigabyte – less than a typical main memory of a laptop. However, there are $\binom{1,000,000}{2}$ or half a trillion pairs of documents. If it takes a microsecond to compute the similarity of two signatures, then it takes almost six days to compute all the similarities on that laptop. □

If our goal is to compute the similarity of every pair, there is nothing we can do to reduce the work, although parallelism can reduce the elapsed time. However, often we want only the most similar pairs or all pairs that are above some lower bound in similarity. If so, then we need to focus our attention only on pairs that are likely to be similar, without investigating every pair. There is a general theory of how to provide such focus, called *locality-sensitive hashing* (LSH) or *near-neighbor search*. In this section we shall consider a specific form of LSH, designed for the particular problem we have been studying: documents, represented by shingle-sets, then minhashed to short signatures. In Section 3.6 we present the general theory of locality-sensitive hashing and a number of applications and related techniques.

3.4.1 LSH for Minhash Signatures

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

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

Example 3.10: Figure 3.6 shows part of a signature matrix of 12 rows divided into four bands of three rows each. The second and fourth of the explicitly shown columns each have the column vector $[0, 2, 1]$ in the first band, so they will definitely hash to the same bucket in the hashing for the first band. Thus, regardless of what those columns look like in the other three bands, this pair of columns will be a candidate pair. It is possible that other columns, such as the first two shown explicitly, will also hash to the same bucket according to the hashing of the first band. However, since their column vectors are different, $[1, 3, 0]$ and $[0, 2, 1]$, and there are many buckets for each hashing, we expect the chances of an accidental collision to be very small. We shall normally assume that two vectors hash to the same bucket if and only if they are identical.

Two columns that do not agree in band 1 have three other chances to become a candidate pair; they might be identical in any one of these other bands.

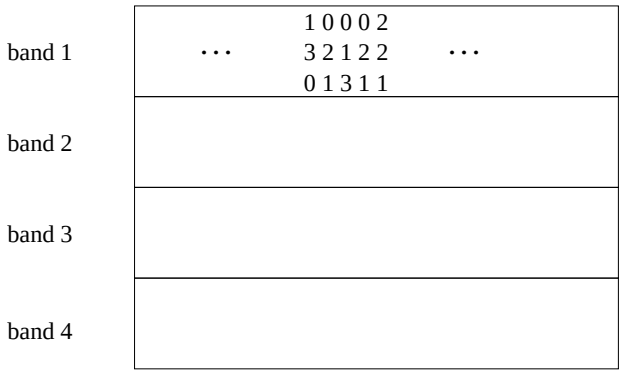

Figure 3.6: Dividing a signature matrix into four bands of three rows per band

However, observe that the more similar two columns are, the more likely it is that they will be identical in some band. Thus, intuitively the banding strategy makes similar columns much more likely to be candidate pairs than dissimilar pairs. □

3.4.2 Analysis of the Banding Technique

Suppose we use b bands of r rows each, and suppose that a particular pair of documents have Jaccard similarity s. Recall from Section 3.3.3 that the probability the minhash signatures for these documents agree in any one particular row of the signature matrix is s. We can calculate the probability that these documents (or rather their signatures) become a candidate pair as follows:

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

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

3. The probability that the signatures disagree in at least one row of each of the bands is $(1 - s^r)^b$.

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

It may not be obvious, but regardless of the chosen constants b and r, this function has the form of an *S-curve*, as suggested in Fig. 3.7. The *threshold*, that is, the value of similarity s at which the probability of becoming a candidate is 1/2, is a function of b and r. The threshold is roughly where the rise is the steepest, and for large b and r there we find that pairs with similarity above the threshold are very likely to become candidates, while those below the threshold are unlikely to become candidates – exactly the situation we want.

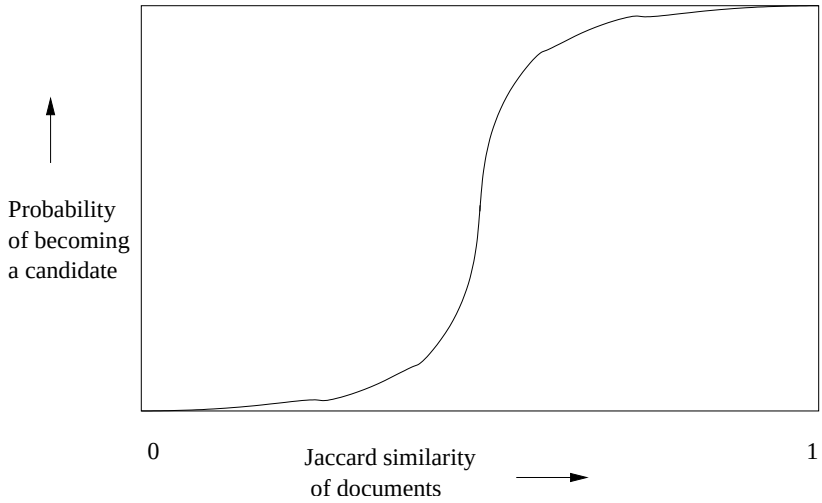

Figure 3.7: The S-curve

An approximation to the threshold is $(1/b)^{1/r}$. For example, if $b = 16$ and $r = 4$, then the threshold is approximately at $s = 1/2$, since the 4th root of $1/16$ is $1/2$.

Example 3.11 : Let us consider the case $b = 20$ and $r = 5$. That is, we suppose we have signatures of length 100, divided into twenty bands of five rows each. Figure 3.8 tabulates some of the values of the function $1 - (1 - s^5)^{20}$. Notice that the threshold, the value of s at which the curve has risen halfway, is just slightly more than 0.5. Also notice that the curve is not exactly the ideal step function that jumps from 0 to 1 at the threshold, but the slope of the curve in the middle is significant. For example, it rises by more than 0.6 going from $s = 0.4$ to $s = 0.6$, so the slope in the middle is greater than 3.

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

Figure 3.8: Values of the S-curve for $b = 20$ and $r = 5$

For example, at $s = 0.8$, $1 - (0.8)^5$ is about 0.672. If you raise this number to the 20th power, you get about 0.00035. Subtracting this fraction from 1

yields 0.99965. That is, if we consider two documents with 80% similarity, then in any one band, they have only about a 33% chance of agreeing in all five rows and thus becoming a candidate pair. However, there are 20 bands and thus 20 chances to become a candidate. Only roughly one in 3000 pairs that are as high as 80% similar will fail to become a candidate pair and thus be a false negative. □

3.4.3 Combining the Techniques

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

1. Pick a value of k and construct from each document the set of k-shingles. Optionally, hash the k-shingles to shorter bucket numbers.
2. Sort the document-shingle pairs to order them by shingle.
3. Pick a length n for the minhash signatures. Feed the sorted list to the algorithm of Section 3.3.5 to compute the minhash signatures for all the documents.
4. Choose a threshold t that defines how similar documents have to be in order for them to be regarded as a desired "similar pair." Pick a number of bands b and a number of rows r such that $br = n$, and the threshold t is approximately $(1/b)^{1/r}$. If avoidance of false negatives is important, you may wish to select b and r to produce a threshold lower than t; if speed is important and you wish to limit false positives, select b and r to produce a higher threshold.
5. Construct candidate pairs by applying the LSH technique of Section 3.4.1.
6. Examine each candidate pair's signatures and determine whether the fraction of components in which they agree is at least t.
7. Optionally, if the signatures are sufficiently similar, go to the documents themselves and check that they are truly similar, rather than documents that, by luck, had similar signatures.

3.4.4 Exercises for Section 3.4

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

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

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

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

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

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

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

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

3.5 Distance Measures

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

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

3.5.1 Definition of a Distance Measure

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

1. $d(x, y) \geq 0$ (no negative distances).

2. $d(x, y) = 0$ if and only if $x = y$ (distances are positive, except for the distance from a point to itself).

3. $d(x, y) = d(y, x)$ (distance is symmetric).

4. $d(x, y) \leq d(x, z) + d(z, y)$ (the *triangle inequality*).

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

3.5.2 Euclidean Distances

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

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

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

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

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

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

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

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

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

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

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

□

3.5.3 Jaccard Distance

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

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

2. $d(x, y) = 0$ if $x = y$, because $x \cup x = x \cap x = x$. However, if $x \neq y$, then the size of $x \cap y$ is strictly less than the size of $x \cup y$, so $d(x, y)$ is strictly positive.

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

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

.

3.5.4 Cosine Distance

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

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

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

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

3.5.5 Edit Distance

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

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

1. Delete `b`.
2. Insert `f` after `c`.

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

3. Insert `g` after `e`.

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

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

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

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

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

3.5.6 Hamming Distance

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

Non-Euclidean Spaces

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

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

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

3.5.7 Exercises for Section 3.5

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

(a) `abcdef` and `bdaefc`.

(b) `abccdabc` and `acbdcab`.

(c) `abcdef` and `baedfc`.

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

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

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

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

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

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

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

Chapter 4

Mining Data Streams

Most of the algorithms described in this book assume that we are mining a database. That is, all our data is available when and if we want it. In this chapter, we shall make another assumption: data arrives in a stream or streams, and if it is not processed immediately or stored, then it is lost forever. Moreover, we shall assume that the data arrives so rapidly that it is not feasible to store it all in active storage (i.e., in a conventional database), and then interact with it at the time of our choosing.

The algorithms for processing streams each involve summarization of the stream in some way. We shall start by considering how to make a useful sample of a stream and how to filter a stream to eliminate most of the "undesirable" elements. We then show how to estimate the number of different elements in a stream using much less storage than would be required if we listed all the elements we have seen.

Another approach to summarizing a stream is to look at only a fixed-length "window" consisting of the last n elements for some (typically large) n. We then query the window as if it were a relation in a database. If there are many streams and/or n is large, we may not be able to store the entire window for every stream, so we need to summarize even the windows. We address the fundamental problem of maintaining an approximate count on the number of 1's in the window of a bit stream, while using much less space than would be needed to store the entire window itself. This technique generalizes to approximating various kinds of sums.

4.1 The Stream Data Model

Let us begin by discussing the elements of streams and stream processing. We explain the difference between streams and databases and the special problems that arise when dealing with streams. Some typical applications where the stream model applies will be examined.

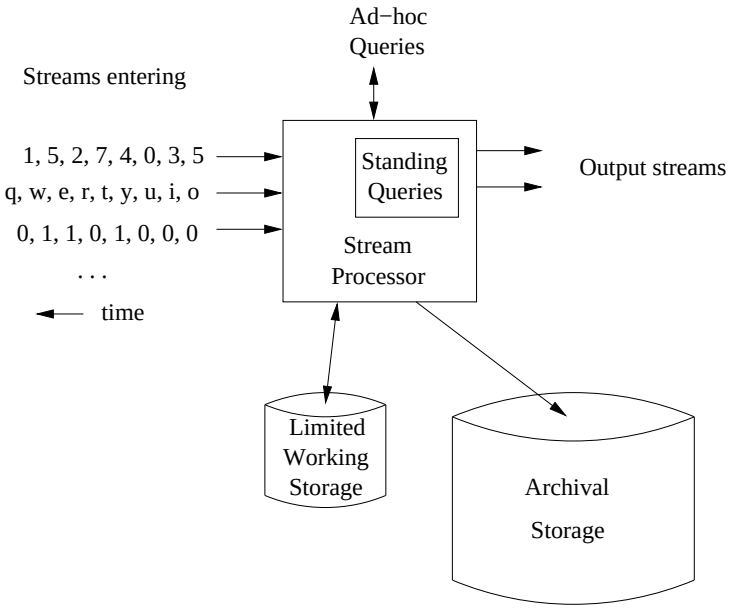

Figure 4.1: A data-stream-management system

4.1.1 A Data-Stream-Management System

In analogy to a database-management system, we can view a stream processor as a kind of data-management system, the high-level organization of which is suggested in Fig. 4.1. Any number of streams can enter the system. Each stream can provide elements at its own schedule; they need not have the same data rates or data types, and the time between elements of one stream need not be uniform. The fact that the rate of arrival of stream elements is not under the control of the system distinguishes stream processing from the processing of data that goes on within a database-management system. The latter system controls the rate at which data is read from the disk, and therefore never has to worry about data getting lost as it attempts to execute queries.

Streams may be archived in a large *archival store*, but we assume it is not possible to answer queries from the archival store. It could be examined only under special circumstances using time-consuming retrieval processes. There is also a *working store*, into which summaries or parts of streams may be placed, and which can be used for answering queries. The working store might be disk, or it might be main memory, depending on how fast we need to process queries. But either way, it is of sufficiently limited capacity that it cannot store all the data from all the streams.

4.1.2 Examples of Stream Sources

Before proceeding, let us consider some of the ways in which stream data arises naturally.

Sensor Data

Imagine a temperature sensor bobbing about in the ocean, sending back to a base station a reading of the surface temperature each hour. The data produced by this sensor is a stream of real numbers. It is not a very interesting stream, since the data rate is so low. It would not stress modern technology, and the entire stream could be kept in main memory, essentially forever.

Now, give the sensor a GPS unit, and let it report surface height instead of temperature. The surface height varies quite rapidly compared with temperature, so we might have the sensor send back a reading every tenth of a second. If it sends a 4-byte real number each time, then it produces 3.5 megabytes per day. It will still take some time to fill up main memory, let alone a single disk.

But one sensor might not be that interesting. To learn something about ocean behavior, we might want to deploy a million sensors, each sending back a stream, at the rate of ten per second. A million sensors isn't very many; there would be one for every 150 square miles of ocean. Now we have 3.5 terabytes arriving every day, and we definitely need to think about what can be kept in working storage and what can only be archived.

Image Data

Satellites often send down to earth streams consisting of many terabytes of images per day. Surveillance cameras produce images with lower resolution than satellites, but there can be many of them, each producing a stream of images at intervals like one second. London is said to have six million such cameras, each producing a stream.

Internet and Web Traffic

A switching node in the middle of the Internet receives streams of IP packets from many inputs and routes them to its outputs. Normally, the job of the switch is to transmit data and not to retain it or query it. But there is a tendency to put more capability into the switch, e.g., the ability to detect denial-of-service attacks or the ability to reroute packets based on information about congestion in the network.

Web sites receive streams of various types. For example, Google receives several hundred million search queries per day. Yahoo! accepts billions of "clicks" per day on its various sites. Many interesting things can be learned from these streams. For example, an increase in queries like "sore throat" enables us to track the spread of viruses. A sudden increase in the click rate for a link could

indicate some news connected to that page, or it could mean that the link is broken and needs to be repaired.

4.1.3 Stream Queries

There are two ways that queries get asked about streams. We show in Fig. 4.1 a place within the processor where *standing queries* are stored. These queries are, in a sense, permanently executing, and produce outputs at appropriate times.

Example 4.1 : The stream produced by the ocean-surface-temperature sensor mentioned at the beginning of Section 4.1.2 might have a standing query to output an alert whenever the temperature exceeds 25 degrees centigrade. This query is easily answered, since it depends only on the most recent stream element.

Alternatively, we might have a standing query that, each time a new reading arrives, produces the average of the 24 most recent readings. That query also can be answered easily, if we store the 24 most recent stream elements. When a new stream element arrives, we can drop from the working store the 25th most recent element, since it will never again be needed (unless there is some other standing query that requires it).

Another query we might ask is the maximum temperature ever recorded by that sensor. We can answer this query by retaining a simple summary: the maximum of all stream elements ever seen. It is not necessary to record the entire stream. When a new stream element arrives, we compare it with the stored maximum, and set the maximum to whichever is larger. We can then answer the query by producing the current value of the maximum. Similarly, if we want the average temperature over all time, we have only to record two values: the number of readings ever sent in the stream and the sum of those readings. We can adjust these values easily each time a new reading arrives, and we can produce their quotient as the answer to the query. □

The other form of query is *ad-hoc*, a question asked once about the current state of a stream or streams. If we do not store all streams in their entirety, as normally we can not, then we cannot expect to answer arbitrary queries about streams. If we have some idea what kind of queries will be asked through the ad-hoc query interface, then we can prepare for them by storing appropriate parts or summaries of streams as in Example 4.1.

If we want the facility to ask a wide variety of ad-hoc queries, a common approach is to store a *sliding window* of each stream in the working store. A sliding window can be the most recent n elements of a stream, for some n, or it can be all the elements that arrived within the last t time units, e.g., one day. If we regard each stream element as a tuple, we can treat the window as a relation and query it with any SQL query. Of course the stream-management system must keep the window fresh, deleting the oldest elements as new ones come in.

Example 4.2: Web sites often like to report the number of unique users over the past month. If we think of each login as a stream element, we can maintain a window that is all logins in the most recent month. We must associate the arrival time with each login, so we know when it no longer belongs to the window. If we think of the window as a relation `Logins(name, time)`, then it is simple to get the number of unique users over the past month. The SQL query is:

```
SELECT COUNT(DISTINCT(name))
FROM Logins
WHERE time >= t;
```

Here, t is a constant that represents the time one month before the current time.

Note that we must be able to maintain the entire stream of logins for the past month in working storage. However, for even the largest sites, that data is not more than a few terabytes, and so surely can be stored on disk. □

4.1.4 Issues in Stream Processing

Before proceeding to discuss algorithms, let us consider the constraints under which we work when dealing with streams. First, streams often deliver elements very rapidly. We must process elements in real time, or we lose the opportunity to process them at all, without accessing the archival storage. Thus, it often is important that the stream-processing algorithm is executed in main memory, without access to secondary storage or with only rare accesses to secondary storage. Moreover, even when streams are "slow," as in the sensor-data example of Section 4.1.2, there may be many such streams. Even if each stream by itself can be processed using a small amount of main memory, the requirements of all the streams together can easily exceed the amount of available main memory.

Thus, many problems about streaming data would be easy to solve if we had enough memory, but become rather hard and require the invention of new techniques in order to execute them at a realistic rate on a machine of realistic size. Here are two generalizations about stream algorithms worth bearing in mind as you read through this chapter:

- Often, it is much more efficient to get an approximate answer to our problem than an exact solution.

- As in Chapter 3, a variety of techniques related to hashing turn out to be useful. Generally, these techniques introduce useful randomness into the algorithm's behavior, in order to produce an approximate answer that is very close to the true result.

4.2 Sampling Data in a Stream

As our first example of managing streaming data, we shall look at extracting reliable samples from a stream. As with many stream algorithms, the "trick" involves using hashing in a somewhat unusual way.

4.2.1 A Motivating Example

The general problem we shall address is selecting a subset of a stream so that we can ask queries about the selected subset and have the answers be statistically representative of the stream as a whole. If we know what queries are to be asked, then there are a number of methods that might work, but we are looking for a technique that will allow ad-hoc queries on the sample. We shall look at a particular problem, from which the general idea will emerge.

Our running example is the following. A search engine receives a stream of queries, and it would like to study the behavior of typical users.[1] We assume the stream consists of tuples (user, query, time). Suppose that we want to answer queries such as "What fraction of the typical user's queries were repeated over the past month?" Assume also that we wish to store only 1/10th of the stream elements.

The obvious approach would be to generate a random number, say an integer from 0 to 9, in response to each search query. Store the tuple if and only if the random number is 0. If we do so, each user has, on average, 1/10th of their queries stored. Statistical fluctuations will introduce some noise into the data, but if users issue many queries, the law of large numbers will assure us that most users will have a fraction quite close to 1/10th of their queries stored.

However, this scheme gives us the wrong answer to the query asking for the average number of duplicate queries for a user. Suppose a user has issued s search queries one time in the past month, d search queries twice, and no search queries more than twice. If we have a 1/10th sample, of queries, we shall see in the sample for that user an expected $s/10$ of the search queries issued once. Of the d search queries issued twice, only $d/100$ will appear twice in the sample; that fraction is d times the probability that both occurrences of the query will be in the 1/10th sample. Of the queries that appear twice in the full stream, $18d/100$ will appear exactly once. To see why, note that 18/100 is the probability that one of the two occurrences will be in the 1/10th of the stream that is selected, while the other is in the 9/10th that is not selected.

The correct answer to the query about the fraction of repeated searches is $d/(s+d)$. However, the answer we shall obtain from the sample is $d/(10s+19d)$. To derive the latter formula, note that $d/100$ appear twice, while $s/10+18d/100$ appear once. Thus, the fraction appearing twice in the sample is $d/100$ divided

[1] While we shall refer to "users," the search engine really receives IP addresses from which the search query was issued. We shall assume that these IP addresses identify unique users, which is approximately true, but not exactly true.

by $d/100 + s/10 + 18d/100$. This ratio is $d/(10s + 19d)$. For no positive values of s and d is $d/(s + d) = d/(10s + 19d)$.

4.2.2 Obtaining a Representative Sample

The query of Section 4.2.1, like many queries about the statistics of typical users, cannot be answered by taking a sample of each user's search queries. Thus, we must strive to pick 1/10th of the users, and take all their searches for the sample, while taking none of the searches from other users. If we can store a list of all users, and whether or not they are in the sample, then we could do the following. Each time a search query arrives in the stream, we look up the user to see whether or not they are in the sample. If so, we add this search query to the sample, and if not, then not. However, if we have no record of ever having seen this user before, then we generate a random integer between 0 and 9. If the number is 0, we add this user to our list with value "in," and if the number is other than 0, we add the user with the value "out."

That method works as long as we can afford to keep the list of all users and their in/out decision in main memory, because there isn't time to go to disk for every search that arrives. By using a hash function, one can avoid keeping the list of users. That is, we hash each user name to one of ten buckets, 0 through 9. If the user hashes to bucket 0, then accept this search query for the sample, and if not, then not.

Note we do not actually store the user in the bucket; in fact, there is no data in the buckets at all. Effectively, we use the hash function as a random-number generator, with the important property that, when applied to the same user several times, we always get the same "random" number. That is, without storing the in/out decision for any user, we can reconstruct that decision any time a search query by that user arrives.

More generally, we can obtain a sample consisting of any rational fraction a/b of the users by hashing user names to b buckets, 0 through $b - 1$. Add the search query to the sample if the hash value is less than a.

4.2.3 The General Sampling Problem

The running example is typical of the following general problem. Our stream consists of tuples with n components. A subset of the components are the *key* components, on which the selection of the sample will be based. In our running example, there are three components – user, query, and time – of which only *user* is in the key. However, we could also take a sample of queries by making *query* be the key, or even take a sample of user-query pairs by making both those components form the key.

To take a sample of size a/b, we hash the key value for each tuple to b buckets, and accept the tuple for the sample if the hash value is less than a. If the key consists of more than one component, the hash function needs to combine the values for those components to make a single hash-value. The

result will be a sample consisting of all tuples with certain key values. The selected key values will be approximately a/b of all the key values appearing in the stream.

4.2.4 Varying the Sample Size

Often, the sample will grow as more of the stream enters the system. In our running example, we retain all the search queries of the selected 1/10th of the users, forever. As time goes on, more searches for the same users will be accumulated, and new users that are selected for the sample will appear in the stream.

If we have a budget for how many tuples from the stream can be stored as the sample, then the fraction of key values must vary, lowering as time goes on. In order to assure that at all times, the sample consists of all tuples from a subset of the key values, we choose a hash function h from key values to a very large number of values $0, 1, \ldots, B-1$. We maintain a *threshold* t, which initially can be the largest bucket number, $B-1$. At all times, the sample consists of those tuples whose key K satisfies $h(K) \leq t$. New tuples from the stream are added to the sample if and only if they satisfy the same condition.

If the number of stored tuples of the sample exceeds the allotted space, we lower t to $t-1$ and remove from the sample all those tuples whose key K hashes to t. For efficiency, we can lower t by more than 1, and remove the tuples with several of the highest hash values, whenever we need to throw some key values out of the sample. Further efficiency is obtained by maintaining an index on the hash value, so we can find all those tuples whose keys hash to a particular value quickly.

4.2.5 Exercises for Section 4.2

Exercise 4.2.1: Suppose we have a stream of tuples with the schema

Grades(university, courseID, studentID, grade)

Assume universities are unique, but a courseID is unique only within a university (i.e., different universities may have different courses with the same ID, e.g., "CS101") and likewise, studentID's are unique only within a university (different universities may assign the same ID to different students). Suppose we want to answer certain queries approximately from a 1/20th sample of the data. For each of the queries below, indicate how you would construct the sample. That is, tell what the key attributes should be.

(a) For each university, estimate the average number of students in a course.

(b) Estimate the fraction of students who have a GPA of 3.5 or more.

(c) Estimate the fraction of courses where at least half the students got "A."

4.3 Filtering Streams

Another common process on streams is selection, or filtering. We want to accept those tuples in the stream that meet a criterion. Accepted tuples are passed to another process as a stream, while other tuples are dropped. If the selection criterion is a property of the tuple that can be calculated (e.g., the first component is less than 10), then the selection is easy to do. The problem becomes harder when the criterion involves lookup for membership in a set. It is especially hard, when that set is too large to store in main memory. In this section, we shall discuss the technique known as "Bloom filtering" as a way to eliminate most of the tuples that do not meet the criterion.

4.3.1 A Motivating Example

Again let us start with a running example that illustrates the problem and what we can do about it. Suppose we have a set S of one billion allowed email addresses – those that we will allow through because we believe them not to be spam. The stream consists of pairs: an email address and the email itself. Since the typical email address is 20 bytes or more, it is not reasonable to store S in main memory. Thus, we can either use disk accesses to determine whether or not to let through any given stream element, or we can devise a method that requires no more main memory than we have available, and yet will filter most of the undesired stream elements.

Suppose for argument's sake that we have one gigabyte of available main memory. In the technique known as *Bloom filtering*, we use that main memory as a bit array. In this case, we have room for eight billion bits, since one byte equals eight bits. Devise a hash function h from email addresses to eight billion buckets. Hash each member of S to a bit, and set that bit to 1. All other bits of the array remain 0.

Since there are one billion members of S, approximately 1/8th of the bits will be 1. The exact fraction of bits set to 1 will be slightly less than 1/8th, because it is possible that two members of S hash to the same bit. We shall discuss the exact fraction of 1's in Section 4.3.3. When a stream element arrives, we hash its email address. If the bit to which that email address hashes is 1, then we let the email through. But if the email address hashes to a 0, we are certain that the address is not in S, so we can drop this stream element.

Unfortunately, some spam email will get through. Approximately 1/8th of the stream elements whose email address is not in S will happen to hash to a bit whose value is 1 and will be let through. Nevertheless, since the majority of emails are spam (about 80% according to some reports), eliminating 7/8th of the spam is a significant benefit. Moreover, if we want to eliminate every spam, we need only check for membership in S those good and bad emails that get through the filter. Those checks will require the use of secondary memory to access S itself. There are also other options, as we shall see when we study the general Bloom-filtering technique. As a simple example, we could use a cascade

of filters, each of which would eliminate 7/8th of the remaining spam.

4.3.2 The Bloom Filter

A *Bloom filter* consists of:

1. An array of n bits, initially all 0's.
2. A collection of hash functions $h_1, h_2, \ldots, h_k$. Each hash function maps "key" values to n buckets, corresponding to the n bits of the bit-array.
3. A set S of m key values.

The purpose of the Bloom filter is to allow through all stream elements whose keys are in S, while rejecting most of the stream elements whose keys are not in S.

To initialize the bit array, begin with all bits 0. Take each key value in S and hash it using each of the k hash functions. Set to 1 each bit that is $h_i(K)$ for some hash function h_i and some key value K in S.

To test a key K that arrives in the stream, check that all of

$$h_1(K), h_2(K), \ldots, h_k(K)$$

are 1's in the bit-array. If all are 1's, then let the stream element through. If one or more of these bits are 0, then K could not be in S, so reject the stream element.

4.3.3 Analysis of Bloom Filtering

If a key value is in S, then the element will surely pass through the Bloom filter. However, if the key value is not in S, it might still pass. We need to understand how to calculate the probability of a *false positive*, as a function of n, the bit-array length, m the number of members of S, and k, the number of hash functions.

The model to use is throwing darts at targets. Suppose we have x targets and y darts. Any dart is equally likely to hit any target. After throwing the darts, how many targets can we expect to be hit at least once? The analysis is similar to the analysis in Section 3.4.2, and goes as follows:

- The probability that a given dart will not hit a given target is $(x-1)/x$.
- The probability that none of the y darts will hit a given target is $\left(\frac{x-1}{x}\right)^y$. We can write this expression as $(1-\frac{1}{x})^{x(\frac{y}{x})}$.
- Using the approximation $(1-\epsilon)^{1/\epsilon} = 1/e$ for small ϵ (recall Section 1.3.5), we conclude that the probability that none of the y darts hit a given target is $e^{-y/x}$.

Example 4.3: Consider the running example of Section 4.3.1. We can use the above calculation to get the true expected number of 1's in the bit array. Think of each bit as a target, and each member of S as a dart. Then the probability that a given bit will be 1 is the probability that the corresponding target will be hit by one or more darts. Since there are one billion members of S, we have $y = 10^9$ darts. As there are eight billion bits, there are $x = 8 \times 10^9$ targets. Thus, the probability that a given target is not hit is $e^{-y/x} = e^{-1/8}$ and the probability that it *is* hit is $1 - e^{-1/8}$. That quantity is about 0.1175. In Section 4.3.1 we suggested that $1/8 = 0.125$ is a good approximation, which it is, but now we have the exact calculation. □

We can apply the rule to the more general situation, where set S has m members, the array has n bits, and there are k hash functions. The number of targets is $x = n$, and the number of darts is $y = km$. Thus, the probability that a bit remains 0 is $e^{-km/n}$. We want the fraction of 0 bits to be fairly large, or else the probability that a nonmember of S will hash at least once to a 0 becomes too small, and there are too many false positives. For example, we might choose k, the number of hash functions to be n/m or less. Then the probability of a 0 is at least e^{-1} or 37%. In general, the probability of a false positive is the probability of a 1 bit, which is $1 - e^{-km/n}$, raised to the kth power, i.e., $(1 - e^{-km/n})^k$.

Example 4.4: In Example 4.3 we found that the fraction of 1's in the array of our running example is 0.1175, and this fraction is also the probability of a false positive. That is, a nonmember of S will pass through the filter if it hashes to a 1, and the probability of it doing so is 0.1175.

Suppose we used the same S and the same array, but used two different hash functions. This situation corresponds to throwing two billion darts at eight billion targets, and the probability that a bit remains 0 is $e^{-1/4}$. In order to be a false positive, a nonmember of S must hash twice to bits that are 1, and this probability is $(1 - e^{-1/4})^2$, or approximately 0.0493. Thus, adding a second hash function for our running example is an improvement, reducing the false-positive rate from 0.1175 to 0.0493. □

4.3.4 Exercises for Section 4.3

Exercise 4.3.1: For the situation of our running example (8 billion bits, 1 billion members of the set S), calculate the false-positive rate if we use three hash functions? What if we use four hash functions?

! **Exercise 4.3.2:** Suppose we have n bits of memory available, and our set S has m members. Instead of using k hash functions, we could divide the n bits into k arrays, and hash once to each array. As a function of n, m, and k, what is the probability of a false positive? How does it compare with using k hash functions into a single array?