Algorithms for Massive Datasets
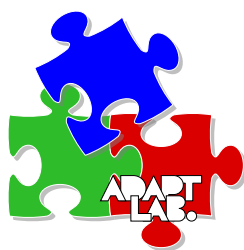
Course held by Prof. Dario Malchiodi

Federico Cristiano Bruzzone
Id. Number: 27427A

MSc in Computer Science

UNIVERSITY OF MILAN
Computer Science Department
ADAPT-Lab

Academic Year 2022–2023

Contents

Contents

List of Figures

1
Data Mining

1.1 Things Useful to Know

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

1.1.1 Importance of Words in Documents

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 formal measure of how concentrated into relatively few documents are the occurrences of a given word is called **TF.IDF**. It is normally computed as follows. Suppose we have a collection of N documents. Define f_{ij} to be the *frequency* 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 the maximum number of occurrences of any term (perhaps excluding stop words) in the same 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 $TF_{ij} \times IDF_i$.

1.1.2 Hash Functions

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.

1.1.3 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.

1.1.4 Secondary Storage

Disks are organized into *blocks*, which are the minimum units that the operating system uses to move data between main memory and disk. It takes approximately ten milliseconds to *access* 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. You can assume that a disk cannot transfer data to main memory at more than a hundred million bytes per second (100MB), 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.

1.1.5 The Base of Natural Logarithms

The constant $e = 2.7182818...$ has a number of useful special properties. In particular:

$$\lim_{n \to \infty} \left(1 + \frac{1}{n}\right)^n = e$$

Some algebra lets us obtain approximations to many complex expression. Condiser $(1 + \alpha)^\beta$, where α a small. We can rewirte the expression as $(1 + \alpha)^{(1/\alpha)(\alpha\beta)}$. Then subsitute $\alpha = 1/n$ and $1/\alpha = n$, so we have that $(1 + \frac{1}{n})^{n(\alpha\beta)}$, which is

$$\left(\left(1 + \frac{1}{n}\right)^n\right)^{\alpha\beta}$$

Since α is assumed small, n is large, so the subexpression $\left(1 + \frac{1}{n}\right)^n$ will be cose to the limiting value of e.

Some other usefull approximations follow from the Taylor expansion of e^x. That is,

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

$$= 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

When x is large, the above series converges slowly, although it does converge because $n!$ grows faster than x^n for any constant x.

1.1.6 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 1.1 suggest such a relationship that is: $\log_{10} y = 6 - 2 \log_{10} x$
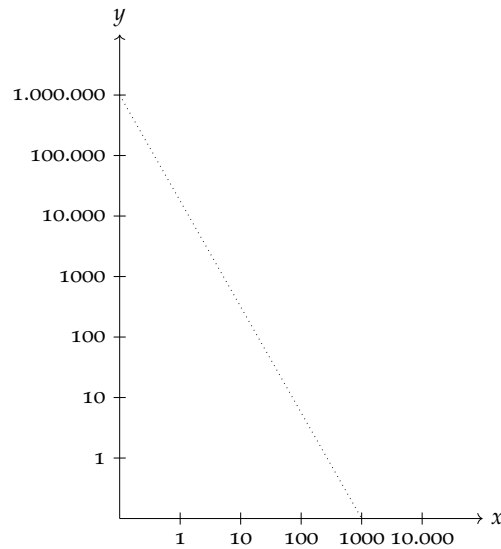
Figure 1.1. *A power law with a slope of* -2.

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.

2

MapReduce and Cost Model

2.1 Algorithms Using MapReduce

MapReduce is not a solution to every problem, not even every problem that profitably can use many compute nodes operating in parallel. The original purpose for which the Google implementation of MapReduce was created was executed very large matrix-verctor multiplication as are needed in the calculation of PageRank (See Chapter 3). We shall see that matrix-vector and matrix-matrix calculations fit nicely into the MapReduce style of computing. Another imortant class of operations that can use MapReduce effectively are the relational-algebra operations.

2.1.1 Matrix-Vector Multiplication by MapReduce

Suppose we have an $n \times n$ matrix M, whose element in row i and column j will be denoted m_{ij}. Suppose we also have a vector $\mathbf{v}$ of length n, whose j-th element is $\mathbf{v}_j$. Then the matrix-vector product is the vector $\mathbf{x}$ of length n, whose i-th element $\mathbf{v}_i$ is given by

$$\mathbf{x}_i = \sum_{j=1}^{n} m_{ij}\mathbf{v}_j$$

Let n be large, but not so large that vector $\mathbf{v}$ cannot fit in memory and thus be avaiable to every Map task.The matrix M and vector $\mathbf{v}$ are stored in the distributed file system (DFS). We assume that the row-column coordinates of each matrix element will be discoverable from from its position in the file, or because it is stored explicitly as a triple (i, j, m_{ij}).We also assume the position of the element $\mathbf{v}_j$ in the vector $\mathbf{v}$ is discoverable in the same way.

   The Map Function: The Map function is written to apply to one element of m. Each Map task will operate on a chunk of the matrix M. From each matrix element m_{ij} it produce a key-value pair $(i, m_{ij}\mathbf{v}_j)$

   The Reduce Function: The Reduce function simply sumus all the values associated with a given key i. The result will be a pair $(i, \mathbf{x}_i)$.

2.1.2 If the Vector v Cannot Fit in Memory

However, it is possible that the vector $\mathbf{v}$ is so large that it will not fit in its entirety in main memory. In this case, we can divide the matrix into vertical *stripes* of equal width

and divide the vector into an equal number of horizontal stripes, of the same height.

$$M \qquad \mathbf{v}$$
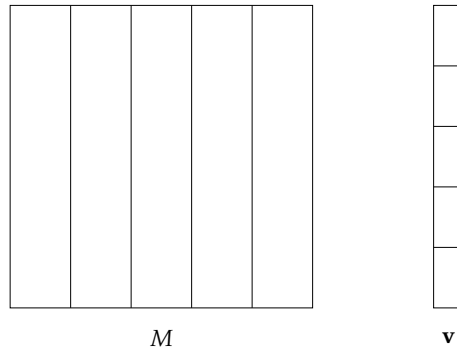
Figure 2.1. *Division of a matrix and vector into stripes.*

Each Map task is assigned a chunk from one of the stripes of the matrix and gets the entire corresponding stripe of the vector. The Map and Reduce tasks can then act exactly as was described above for the case where Map tasks get the entire vector. We shall take up matrix-vector multiplication using MapReduce again in Chapter 3.

2.1.3 Relational-Algebra Operations

A good starting point for exploring applications of MapReduce is by considering the standard operations on relations. A *relation* is a table with column headers called *attributes*, rows of the relation are called *tuples* and the sets of attributes of a relation is called its *schema*.

We often write an expression like $R(A_1, A_2, \ldots, A_n)$ to say that a relation R has the attributes $(A_1, A_2, \ldots, A_n)$

From	To
url1	url2
url1	url3
url2	url3
url3	url4
...	...

Figure 2.2. *Relation Links consists of the set of pairs of URL's, one or more links.*

A relation can be stored as a file in a distributed file system. There are several standard operations on relations, often referred to as *reletional algebra*, that used to implement queries. The relational-algebra operations are:

Relation (represents a table): $R(A, B) \subseteq A \times B$

1. *Selection*: Apply a condition C to each tuple in the relation and produce as output

only those tuple that satisfy C. The result of this selection is denoted $\sigma_C(R)$.

$$\forall t \in R \xrightarrow{MAP} (t,t) \text{ if } c(t)$$
$$(t,(t)) \xrightarrow{REDUCE} (t,t)$$

Map: For each tuple t in R, test if it satisfies C. If so, produce the key-value pair (t,t).
Reduce: is the identity.

2. *Projection*: For some subset S of the attributes of the relation, produce from each tuple only the components for the attributes in S. The result of this projection is denoted $\pi_S(R)$.

$$\forall t \in R \xrightarrow{MAP} (t',t')$$
$$(t',(t',\ldots,t')) \xrightarrow{REDUCE} (t',t')$$

Map: create a tuple t' that contains only the attributes in S.
Reduce: for each key t' produced by the Map, there will be ore or more key-value pairs (t',t'), so the Reduce function will remove the duplicates.

3. *Union*, *Intersection*, and *Difference*: These operations are defined in the usual way.

Union:
$$\begin{cases} \forall t \in R \xrightarrow{MAP} (t,t), \quad \forall t \in S \xrightarrow{MAP} (t,t) \\ \qquad (t,(t)) \xrightarrow{REDUCE} (t,t) \\ \qquad (t,(t,t)) \xrightarrow{REDUCE} (t,t) \end{cases}$$

Intersection:
$$\begin{cases} \forall t \in R \xrightarrow{MAP} (t,t), \quad \forall t \in S \xrightarrow{MAP} (t,t) \\ \qquad (t,(t)) \xrightarrow{REDUCE} \varnothing \\ \qquad (t,(t,t)) \xrightarrow{REDUCE} (t,t) \end{cases}$$

Difference:
$$\begin{cases} \forall t \in R \xrightarrow{MAP} (t,'R'), \quad \forall t \in S \xrightarrow{MAP} (t,'S') \\ \qquad (t,R) \xrightarrow{REDUCE} (t,t) \\ \qquad (t,S) \xrightarrow{REDUCE} \varnothing \\ \qquad (t,(R,S)) \xrightarrow{REDUCE} \varnothing \end{cases}$$

Considering the Union:

– **Map**: Turn each input t ubti a key-value pair (t,t)
– **Reduce**: Associated with each key t there will be either one or two values. Produce output (t,t) ub either case.

Considering the Intersection:

- **Map**: Turn each input t ubti a key-value pair (t, t)
- **Reduce**: If key t has value list (t, t), then produce (t, t), otherwise produce nothing.

Considering the Intersection:

- **Map**: For a tuple t in R, produce the key-value pair (t', R'). For a tuple t in S, produce the key-value pair $(t,' S')$.
- **Reduce**: For each key t, if the associated value list is (t', R'), then produce (t, t), otherwise produce nothing.

4. *Natural Join* : Given two realtions, compare each pair of tuples, one from each relation. If the tuples agree on all the attributes that are common to the two schemas, then produce a tuple that has components for each of the attributes in either schema and agrees with the two tuples on each attribute. If the tuples disagree on one or more shared attributes, then produce nothing from this pair of tuples. All *equijoins* can be executed in the same manner. The natural join of $R(A, B)$ and $S(B, C)$ is denoted $R \bowtie S$.

$$\forall (a, b) \in R \xrightarrow{MAP_R} (b, (a,' R'))$$

$$\forall (b, c) \in S \xrightarrow{MAP_S} (b, (c,' S'))$$

$$(b, ((a_1, R), (c_1, S), (c_3, S), (a_2, R), \dots)) \xrightarrow{REDUCE} \begin{cases} \text{1. Sort } \ell \text{ using 2nd element} \\ \text{2. } a \text{ list of elements from } R \\ \text{3. } b \text{ list of elements from } S \\ \text{4. } \forall (a, c) \in a \times b \\ \text{5. Output}(a, b, c) \end{cases}$$

Map: For each tuple (a, b) of R, produce the key-value pair $(b, (a', R'))$. For each tuple (b, c) of S, produce the key-value pair $(b, (c', S'))$.

Reduce: Each key value b will be associated with a list of pairs that are either of the form (R, a) or (S, c). Construct all pairs consisting of one with first component R and the other with first component S. The output from this key and value list is a sequence of key-value pairs. Each value is one of the triple (a, b, c) such that (R, a) and (S, c) are in the value list.

5. *Grouping* and *Aggregation*: Given a relation R, partition its tuples according to their values in one set of attributes G, called the *grouping attributes*. The permitted aggregations are SUM, COUNT, AVG, MIN, and MAX, with the obvious meanings. We denote a grouping-and-aggregation operation on a relation R by $\gamma_X(R)$, where X is a list of element that are either.

   (a) A grouping attribute, or
   (b) An expression $\theta(A)$, where A is an attribute and θ is an aggregation function.

The result of this operation is one tuple for each group, with components for each grouping attribute and for each aggregation.

$$\forall (a,b) \in R \xrightarrow{MAP} (a,b)$$

$$(a,(b_1,\ldots,b_n)) \xrightarrow{REDUCE} (a,\theta(b_1,\ldots,b_n))$$

Map: For each tuple (a,b,c) produce the key-value pair (a,b).
Reduce: Each key a represent a group. Apply the aggregation operator θ to the list $(b_1,\ldots,b_n)$ of values associated with a. The output is the pair (a,x), where x is the result of the aggregation.

2.1.4 Matrix Multiplication

If M is a matrix with element m_{ij} in row i and column j, and N is a matrix with element n_{jk} in row j and column k, then the product $P = MN$ is the matrix P with element p_{ik} in row i and column k, where

$$p_{ik} = \sum_j m_{ij} n_{jk}$$

Obviously, the number of columns of M must be equal to the number of rows of N, so the sum over j makes sense. We could view matrix M and N as relations, $M(I,J,V)$ with tuples (i,j,m_{ij}) and $N(J,K,V)$ with tuples (j,k,n_{jk}). The product MN is a natural join, $M(I,J,V) \bowtie N(J,K,V)$, having only attribute J in common and produce tuples (i,j,k,v,w) for each tuple (i,j,v) in M and each tuple (j,k,w) in N. This five-component tuple represents the pair of matrix element (m_{ij},n_{ik}) but we prefer to have a four-component tuple $(i,j,k,v \times w)$ because represents the product between $m_{ij}n_{jk}$.

We can perform grouping using I and K as attributes and aggregation as the sum of $V \times W$, and we can implement matrix multiplication as two MapReduce operation. Let N_c the number of columns of N and M_r the number of rows of M.

$$M = [m_{ij}]_{M_r \times X} \qquad N = [n_{jk}]_{X \times N_c} \qquad P = MN = [p_{ik}]_{M_r \times N_c} \qquad p_{ik} = \sum_{j=1}^{X} m_{ij} n_{jk}$$

$$(i,j,m_{ij}) \in M(I,J,V)$$

$$(j,k,n_{jk}) \in N(J,K,W)$$

$$M \bowtie N(i,j,k,m_{ij},n_{jk})$$

First implementation:

$$\forall (i,j,m_{ij}) \in M \xrightarrow{Map_M} (j,(M,i,m_{ik}))$$

$$\forall (j,k,n_{jk}) \in N \xrightarrow{Map_N} (j,(N,k,n_{jk}))$$

$$(j,((M,1,m_{1j}),\ldots,(M,M_r,m_{M_rj})),(N,1,n_{j1}),\ldots,(N,N_c,n_{jN_c})))$$

Map: For each matrix element m_{ij}, produce the key value pair $(j,(M,i,m_{ij}))$ and n_{jk} produce the key value pair $(j,(N,k,n_{jk}))$. M and N in the values are not the matrices but just labels to distinguish the two sources of tuples.

Reduce: For each key j, look at the list of values. For each value that comes from M and N, produce the key-value pair with key equal to (i,k) and value equal to the product of these elements $m_{ij}n_{jk}$.

Now, we perform a grouping and aggregation by another MapReduce operation.

$$((i,j),(a_{ik}b_{kj})) \xrightarrow{Map} ((i,j),(a_{ik}b_{kj}))$$

$$((i,j),(a_{i1}b_{1j})) \xrightarrow{Reduce} ((i,j),p_{ij}) \text{ where } p_{ij} \in P$$

Map: This function is just the identity.

Reduce: For each key (i,k), producethe sum of the list of values associated with this key. The result is the key-value pair with key (i,k) and value equal to the product p_{ik} where p_{ik} is the value of the matrix $P = MN$ in row i and column k.

Second implementation:

We can perform matrix multiplication $P = MN$ in a **single MapReduce operation**. Let N_c the number of columns of N and M_r the number of rows of M.

$$\forall m_{ij} \in M \xrightarrow{Map_M} ((i,k),(M,j,m_{ij})) \quad \forall k = 1,2,\ldots,N_c$$

$$\forall n_{jk} \in N \xrightarrow{Map_N} ((i,k),(N,j,n_{jk})) \quad \forall i = 1,2,\ldots,M_r$$

$$((i,k),((M,1,m_{i1}),\ldots,(M,X,m_{iX}),(N,1,n_{1j}),\ldots,(N,X,n_{Xk})) \xrightarrow{Reduce} ((i,k),p_{ik})$$

Map: For each matrix element m_{ij}, produce the key value pair $((i,k),(M,j,m_{ij}))$ and for each matrix element n_{jk}, produce the key value pair $((i,k),(N,j,n_{jk}))$.

Reduce: Each key (i,k) have an associated list of values $((M,j,m_{ij}),(N,j,n_{jk})) \forall j$. The reduce function needs to connect the two values with the same j, an easy way to do this step is sort by j in two separate lists and the j-th element will have the third componenet m_{ij} and n_{jk}, then we can compute the product $m_{ij}n_{jk}$ and sum over j.

2.2 The communication cost model

The communication cost model is a model that allows us to estimate the communication cost of a MapReduce algorithm. The communication cost is the amount of data that

is transferred between the mappers and the reducers. The communication cost is the most important factor in the performance of a MapReduce algorithm.

Imagine that an algorithm is implemented by an acyclic network of tasks. The *communication cost* of a task is the size of the input to the task and the task can be measured in bytes. The *communication cost of an algorithm* is the sum of the communication cost of all the tasks implementing that algorithm.

We can explain and jusity the **importance of communication** cost as follows:

– The algorithm execution by each task tends to be very simple.
– The typical interconnect speed for a computing cluster is 1 Gbps, and it is slow compared to the speed of the CPU.
– Even if a task executes at a compute node that has a copy of the chunk(s) on which the task operates, that chunk normally will be stored on disk, and the time taken to move the data into main memory may exceed the time needed to operate on the data once it is available in memory.

We might still ask why we **count only input size**, and not output size. The answare to this question involves two points:

1. If the output of the one task τ is the input of another task, then the size of τ's output will be accounted for when measuring the input size for the receiving task.
2. The output is rarely large compared with the output with the input or the intermediate data produced by the algorithm.

An example of join operation:

$$R(A, B) \bowtie S(B, C)$$
$$\forall (a, b) \in R \xrightarrow{M} (b, (a, R))$$
$$\forall (b, c) \in S \xrightarrow{M} (b, (c, S))$$
$$|R| = r, \qquad |S| = s$$
$$\text{Cost: } O(r + s)$$

2.2.1 Wall-Clock Time

While communication cost often influences our choice of algorithm to use in a cluster-computing environment, we must also be aware of the importance of *wall-clock time*, the time it takes a parallel algorithm to finish.

2.2.2 Multiway Joins

To see how analyzing the communication cost can help us choose an algorithm in the cluster-computing environment, we can consider the problem of computing a multiway join. There is a general theory in which we:

1. Select certain attributes of the realtions involved in the natural *multi*-join to have their value hashed, each to some number of buckets.

2. Select the number of buckets for each of these attributes, subejct to the contraint that the product of the number for each attribute is k.

3. Idetify each of the k reducers with a vector of bucket numbers. These vectors have one component for each of the attributes selected in step (1).

4. Send tuples of each realtion to all those reducers where it might find tuples to join with. The given tuple t will have values for some of the attributes selected at step (1), in this way we can apply the hash function(s) to those values to determine certain components of the vector that identifies the reducers.

Cosider the following example $R(A, B) \bowtie S(B, C) \bowtie T(C, D)$ and suppose that R, S, and T have sizes r, s, and t, respectively. And let p be the probability that:

- An R-tuple and an S-tuple have the same value for attribute B.
- An S-tuple and a T-tuple have the same value for attribute C.

If we join R and S first (see 4), then the communication cost is $O(r + s)$ and the size of the intermediate join is $R \bowtie S$ is prs. When we join this result with T, the communication cost is $O(prs + t)$. The total communication cost is $O(r + s + prs + t)$.

A three way to take this join is to use a single MapReduce job that joins the three relation at one time. Suppose that we have k reducers for this job. Pick numbers b and c representing the number of buckets into which we will hash B- and C-values, respectively. Let h and g be a hash function defined as follows:

$$h : B \rightarrow \{0, 1, \ldots, b - 1\}$$
$$g : C \rightarrow \{0, 1, \ldots, c - 1\}$$

We require that $bc = k$. The reducer corresponding to bucket pair (i, j) is responsible for joining the tuples $R(u, v)$, $S(v, w)$, and $T(w, x)$, where $h(v) = i$ and $g(w) = j$.

As a result, the Map tasks that send tuples of R, S and T to the reducers that need them must send R- and T-tuples to more than one reducer. For an S-tuple $S(v, w)$, we know B and C values, so we can compute $h(v)$ and $g(w)$ and send the tuple to the reducer corresponding to bucket pair $(h(v), g(w))$. If we consider the R- and T-tuples, we know only the B- and C-values, respectively, so we can send the tuple to all reducers corresponding to bucket pairs $(h(v), j)$ and $(i, g(w))$, respectively.
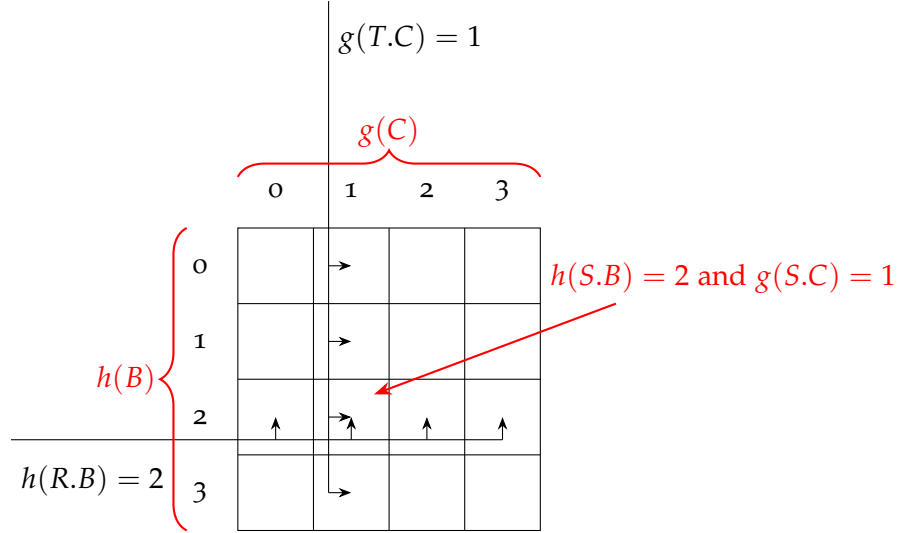
Figure 2.3. *Sixteen reducers together perform a 3-way join.*

Notice that these three tuples join, and they meet at exactly one reducer, the reducer for key (2, 1).

Now, suppose that the sizes of R, S, and T are different, and let r, s, and t be the sizes of R, S, and T, respectively. If we hash B-values into b buckets and C-values into c buckets, where $bc = k$, then the total communication cost to send tuples to the reducers is $O(r + s + t)$. We shall use the of Lagrangean multipliers to find the palce where the function $s + cr + bt - \lambda(bc - k)$ has derivatives with respect to b and c equal to 0. That is, we must solve the equations $r - \lambda c = 0$ and $t - \lambda b = 0$. Since $r = \lambda b$ and $t = \lambda c$. Since $r = \lambda b$ and $t = \lambda c$, we may multiply corresponding sides of these equations to get $rt = \lambda^2 bc$. Since $bc = k$, we get $rt = \lambda^2 k$, or $\lambda = \sqrt{rt/k}$. Thus, $b = \sqrt{r/k}$ and $c = \sqrt{t/k}$.

If we substitute these values into the formula $s + cr + bt$, we get $s + 2\sqrt{rtk}$. This is the communication cost for the Reduce tasks, to which we must add the communication cost for the Map tasks, which is $O(r + s + t)$. Thus, the total communication cost is $O(r + 2s + t + 2\sqrt{rtk})$.

3

Link Analysis

3.1 Definition of PageRank

PageRank is a function that assigns a real number to each page in the Web. There is not one fixed algorithm for assignment of PageRank.

In general, we can define the *transition matrix of the Web* to describe what happens to random surfers after one step. This matrix $M_{n \times n}$ where n is the number of pages. The element m_{ij} has value $1/k$ if page j has k arcs out, and one of them is to page i. Otherwise, $m_{ij} = 0$.

The probability distribution for the location of a random surfer can be described by a column vector whose j-th component is the probability that the surfer is at page j. This probability is the (idealized) PageRank function.

Suppose we start a random surfer A at any of the n pages of the Web with equal probability. Then the initial vector $\mathbf{v}_0 = [1/n]_n$, if M is the matrix of the web, the distribution of A is $M^s \mathbf{v}_0$ for $s = 1, 2, \ldots$ steps. To see why multiplying a distribution vector $\mathbf{v}$ by M gives the distribution $\mathbf{x} = M\mathbf{v}$ at the next step. The probability that A will be at node i at the next step, is (ancient theory of *Markov processes*).

$$\mathbb{P}[\mathbf{x}_i] = \sum_j m_{ij} \mathbf{v}_j$$

$$\text{where } m_{ij} = \mathbb{P}[A \text{ goes from } j \text{ to } i \text{ at the next step}]$$

$$\mathbf{v}_j = \mathbb{P}[A \text{ is at } j \text{ at the previous step}]$$

The limiting distribution $\mathbf{v}$ that satisfies $\mathbf{v} = M\mathbf{v}$ is called the *stationary distribution* of the Markov process. It is the distribution that the A will eventually reach, no matter where he starts. In other words, the limiting $\mathbf{v} = \lambda M\mathbf{v}$ is an eigenvector of M with eigenvalue λ. Remember that M is a stochastic matrix, so $\sum_j m_{ij} = 1 \ \forall i$, and the eigenvalue associated to principal eigenvector is $\lambda = 1$. Recall that the intuition behind PageRank is that the more likely a surfer is to be at a page, the more important the page is.

3.2 Structure of the Web

It would very nice if the nice if the Web were strongly connected, it is not in practice. An early study of the Web found it to have the structure shown in Fig. 3.1. There was a

large strongly connected component (SCC), but there were several other portions that were almost as large.

1. *in-component*, consist of pages that can reach the SCC, but cannot be reached from the SCC.
2. *out-component*, consist of pages that can be reached from the SCC, but unable to reach the SCC.
3. *tendrils*, which are of two types. Some tendrils consist of pages reachable from the in-component but not able to reach the in-component. The other tendrils can reach the out-component, but are not reachable from the out-component.
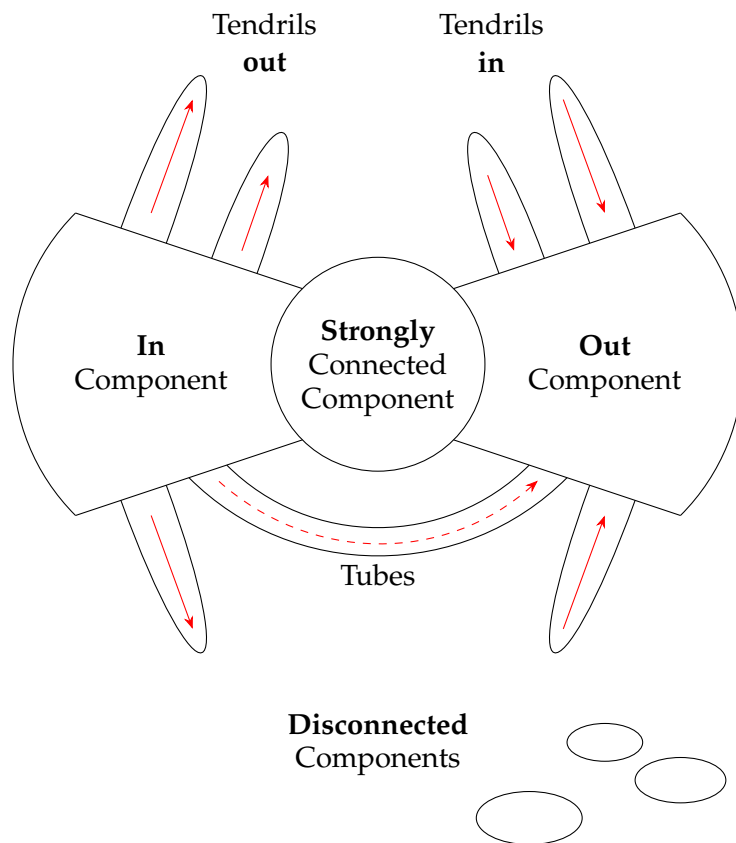
Figure 3.1. *The "bowtie" picture of the Web.*

(a) *Tuples*, which are pages from the in-component and able to reach the out-component, but unable to reach the SCC or be reached from the SCC.
(b) Isolated, components that are unreachable from the large components and unable to reach those components.

There are really two problems we need to avoid. First is the dead end, a page that has no links out. The second problem is groups of pages that all have outlinks but they never link to any other pages.

3.3 Avoiding Dead Ends

Recall that a page with no link out is called a dead end. If we allow dead ends, the transition matrix of the web is no longer stochastic. If we compute $M^i\mathbf{v}$ for increasing powers of a substoachastic matrix M, then some or all of the components of the vector go to 0.

There are two approaches to dealing with dead ends.

1. We can drop the dead ends from the graph, and also drop their incoming arcs. Doing so may create more dead ends, which also have to be dropped, recursively.
2. We can modify the process by which random surfers are assumed to move about the Web. This method, which we refer to as "taxation", also solves the problem of spider traps.

3.3.1 Recursive deletion of dead ends

In order to compute the PageRank with dead ends, using the first method (**drop dead ends**),
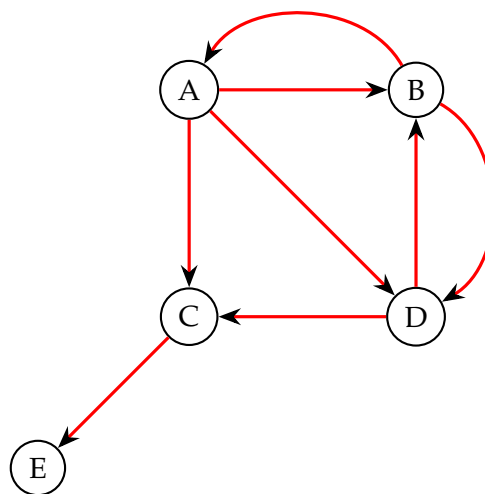
Figure 3.2. *A graph with two levels of dead ends.*

- Drop the dead ends from the graph, and also drop their incoming arcs.
- Compute the PageRank of the remaining nodes, using the method described in the previous section.

– Compute the PageRank of the last removed dead ends x as follows:

$$PR_{dead}(x) = \sum_{v \in G \,:\, (v,x) \in E} PR(v) \frac{1}{\mathbf{deg}(v)}$$

where $\mathbf{deg}(v)$ is the out-degree of v;   (3.1)

G is the graph after removing dead ends;

$PR(v)$ is the PageRank of v in G.

and continue recursively for all removed dead ends.
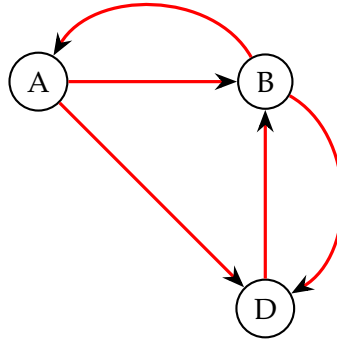
Figure 3.3. *The reduced graph with no dead ends.*

3.3.2 Spider traps and Taxation

A spider trap is a set of nodes with no dead ends but no arcs out. An example of a transition matrix representing the Fig. 3.4 with a single spider trap is:

$$M = \begin{bmatrix} 1 & 1/2 & 0 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 1 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix}$$

The problem in the following example is that the spider trap is absorbing. Once the surfer enters the spider trap, he can never leave.
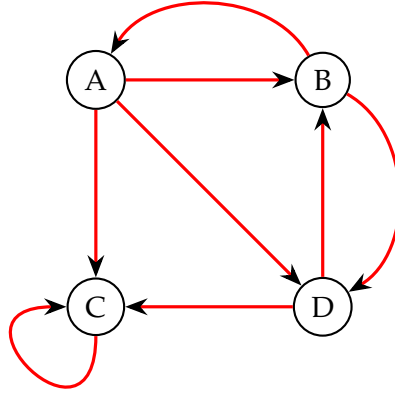
Figure 3.4. *A graph with a one-node spider trap.*

To avoid this problem, we can modify the calculation of PageRank by adding a small probability β that the surfer will jump to a random page. We compute the new vector estimate of PageRanks $\mathbf{v}'$ as follows:

$$\mathbf{v}' = \beta M \mathbf{v} + (1 - \beta)\mathbf{e}/n \tag{3.2}$$

where β is a chosen constant, usually in the range 0.8 to 0.9, $\mathbf{e}$ is a vector of all 1's, and n is the number of nodes in the graph. The term $\beta M \mathbf{v}$ represent the case where, with probability β a random surfer will follow a link out of the current page. The term $(1 - \beta)\mathbf{e}/n$ represents the case where, with probability $1 - \beta$, the surfer will jump to a random page.

3.4 Efficient Computation of PageRank

To compute the PageRank for a large graph representing the Web, we have to perform a matrix-vector multiplication on the order of 50 times, until the vector is close to unchanged at one interation. There are basically two issues:

– The transition matrix of the Web M is very sparse. We want to take to represent the matrix by its non-zero elements.
– We may not be using MapReduce, or for efficient reason we may wish to use a combiner with the Map tasks to reduce the amount of data.

3.4.1 Representing transition matrix

The proper way to represent any sparse matrix is to list the locations of the nonzero entries and their values. We use 4-byte integers for coordinates of an element and an 8-byte double-precision number for the value, so the total size is 16-bytes per nonzero element. The space required is $O(n)$, where n is the number of nonzero elements.

To store a transition matrix using less space, we can thus represent a column by one integer for the out-degree, and one integer per nonzero entry in that column, giving the row number where that entry is located. Thus, we need slightly more than 4 bytes per nonzero entry to represent a transition matrix.

3.4.2 PageRank Iteration Using MapReduce

One iteration of the PageRank algorithm involves taking an estimated PageRank vector $\mathbf{v}$ and computing the next estimate $\mathbf{v}'$ using Eq. 3.2. If n is small enough, we can compute $\mathbf{v}'$ in memory and then write it out to a file. If n is large, as it is for the Web, we can use MapReduce to compute $\mathbf{v}'$ dividing M into vertical stripes and $\mathbf{v}$ into horizontal stripes. This method is not adequate because we are assuming that $\mathbf{v}$ cannot fit in memory, and $\mathbf{v}'$, that is the product between M and $\mathbf{v}$, has the same size as $\mathbf{v}$, so it cannot fit in memory either.

An alternative strategy is based on partitioning M into k^2 blocks, while $\mathbf{v}$ are stil partitioned into k stripes.

$$
\begin{array}{|c|}
\hline \mathbf{v}'_1 \\ \hline \mathbf{v}'_2 \\ \hline \mathbf{v}'_3 \\ \hline \mathbf{v}'_4 \\ \hline
\end{array}
=
\begin{array}{|c|c|c|c|}
\hline M_{11} & M_{12} & M_{13} & M_{14} \\ \hline M_{21} & M_{22} & M_{23} & M_{24} \\ \hline M_{31} & M_{32} & M_{33} & M_{34} \\ \hline M_{41} & M_{42} & M_{43} & M_{44} \\ \hline
\end{array}
\begin{array}{|c|}
\hline \mathbf{v}_1 \\ \hline \mathbf{v}_2 \\ \hline \mathbf{v}_3 \\ \hline \mathbf{v}_4 \\ \hline
\end{array}
$$

Figure 3.5. *Partitioning a matrix into square blocks.*

In this method, we use k^2 Map tasks. Each task gets one block of M denote as M_{ij} and one stripe of the vecor $\mathbf{v}$, which must be $\mathbf{v}_j \ \forall \ k \in i$. Note that $\mathbf{v}$ is trasmitted k times and each block of M only once.

The advantage of this approach is that we can keep the j-th stripe of $\mathbf{v}'$ and $\mathbf{v}$ in memory while we process M_{ij}

Since we are representing transition matrices in the special way described in Section 3.4.1, we need to consider how the block of Fig. 3.5 is represented. For each block, we need data abould all those columns that have at least one nonzero entry within the block. If k is large, then most columns will have nothing in most blocks of its stripe. That, For a given block, we not only have to list those rows that have a nonzero entry for that column, but we must repeat the out-degree for the node represented by the column. That observation upper bounds the space needed to store the blocks of a stipe at twice the space needed to store all the stipe.

3.4.3 Other Efficient Approaches to PageRank Iteration

The algorithm discussed in Section 3.4.2 is not the only way to compute PageRank using MapReduce. Recall that the algorithm of Section 3.4.2 uses k^2 processors, assuming all Map operations are executed in parallel. We can assign all the blocks in one row of blocks to a single Map task, and thus reduce the number of Map tasks to k. We can read the block in a row of blocks one-at-a-time, so the matrix does not consume a significant amount of main-memory. At the same time that we read M_{ij}, we must read the vector stripe $\mathbf{v}_j$. As a result, each of the k Map tasks reads the entire vecotr $\mathbf{v}$, along with $1/k$-th of the matrix.

Source	Degree	Destinations
A	3	B
B	2	A

(a) Representation of M_{11} connecting A and B to B and C

Source	Degree	Destinations
C	1	A
D	2	B

(b) Representation of M_{12} connecting C and D to A and B

Source	Degree	Destinations
A	3	C, D
B	2	D

(c) Representation of M_{21} connecting A and B to C and D

Source	Degree	Destinations
D	2	C

(d) Representation of M_{22} connecting C and D to C and D

Figure 3.6. *Sparse representation of the blocks of a matrix.*

The algorithm is the same as the one described in Section 3.4.2.But the advantage of this approach is that each Map task can combine all the terms for the portion $\mathbf{v}'_i$ for which it is responsible.

3.5 Topic-Sensitive PageRank

We can improve the quality of the results of PageRank weighting certain pages more heavily because of their topic. In the next section, we shall see how the topic-sensitive idea can also be applied to negate the effects of a new kind of spam, called "link spam", that has developed to try to fool the PageRank algorithm. The *topic-sensitive PageRank* approach creates one vector for each of some small number of topics, biasing the PageRank to favor pages of that topic.

3.5.1 Biased Random Walks

Suppose we have identified some pages that represent a topic such as "sports". To create a topic-sensitive PageRank for sports, we can arrange that the random surfers are introduced only to a random sports page, rather than to a random page of any kind. The intuition is that pages linked to by sports pages are themselves likely to be sports pages.

The mathematical formulation for the iteration that yields topic-sensitive PageRank is similar to the equation we used for general PageRank. The only difference is how we add the new surfers. Suppose S is a set of integers consisting of the consisting of the row / column number for the pages we have identified as belonging to a certain topic (called the *teleport set*). Let $\mathbf{e}_S$ be a vector that has 1 in the components in S and 0 in other components. Then the *topic-sensitive PageRank* for S is the limit of the iteration

$$\mathbf{v}' = \beta M\mathbf{v} + (1 - \beta)\frac{\mathbf{e}_S}{|S|}$$

Hence, as usual, M is the transition matrix of the Web, and $|S|$ is the size of set S.

3.5.2 Using Topic-Sensitive PageRank

In order to integrate topic-sensitive PageRank into a search engine, we must:

1. Decide on the topis for which we will create specialized PageRank vectors.
2. Pick a teleport set for each of these topics, and compute the topic-sensitive PageRank vector.
3. Find a way of determining the set of topic that are most relevant.
4. Use the PageRank vectors to order the results of a query.

We have mentioned one way of selecting the topic set: use the top-level topics of the Open Directory.

The third step is probably the trickiest. Some possible approaches are:

(a) Allow the user to specify the topic.
(b) Infer the topic(s) by the words that appear in the Web pages recently serched by the user. See 3.5.3.
(c) Infer the topic(s) by information about the user.

3.5.3 Inferring Topics from Words

Suffice it to say that topics are characterized by words that appear surprisingly often in documents on that topic.

If we examine the entire Web, or a sample, we can get the frquency of each word. In making this judgment, we must be careful to avoid some extremely rare word that appears in the sports sample with relatively higher frequency.

Once we have identified a large collection of words that appear much more frequently for all topics, we can examine other pages and classify them according to the topic that has the highest frequency of the words in the page.

Suppose that $S_1, S_2, \ldots, S_k$ are the sets of words that have been determined to be characteristic of each of the topics on our list. Let P be the set of words that appear in a given page P. Compute the Jaccard similarity between P and each of the sets S_i. Classify P as belonging to the topic S_i for which the Jaccard similarity is largest.

3.6 Link Spam

The techniques for artificially increasing the PageRank of a page are collectively called *link spam*.

3.6.1 Architecture of a Spam Farm
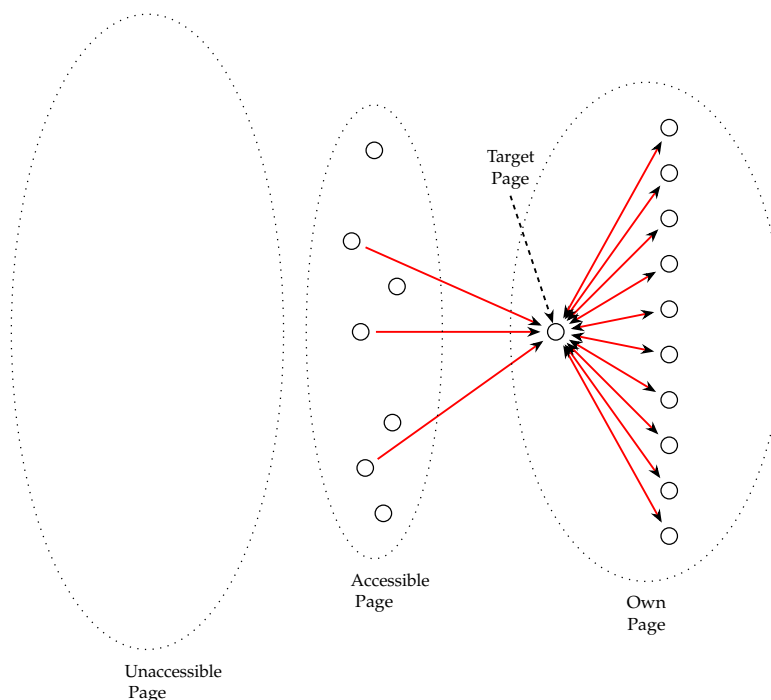
Figure 3.7. *The Web from the point of view of the link spammer.*

A collection of pages purpose is to increas the PageRank of a certain page or pages is called a *spam farm*. Fig. 3.7 show the simplest form of spam farm. From the point of view of the spammers, the Web is divided into three parts:

1. *Inaccessible pages*: most of the Web, which is not accessible to the spammers.
2. *Accessible pages*: accessible to the spammers, but not controlled by them.
3. *Own pages*: pages that are controlled by the spammers.

The spam farm consist of the spammers's own pagas, organized in a special way as seen on the right side of Fig. 3.7 and some links from accessible pages to the spammer's pages. Without some links from the outside, the spam farm would be useless, since it would not even be crawled by a tipical search engine.

In the spam farm, there is a *target page t*, at which the spammer attempts to place as much PageRank as possible. The supporting pages prevent the target page from being an isolated node.

3.6.2 Analysis of a Spam Farm

Suppose that PageRank is computed using a taxation parameter β, that is the fraction of a page's PageRank that is distributed to the pages it links to. Let n be the number of pages in the Web (including m supporting pages and the target page t). Let x (the amount of PageRank of the accessible pages) be the sum over all accessible pages p with a link to t of the PageRank of $p\beta$ divided by the number of successors of p. Finally, let y the unknown PageRank of the target page t.

The PageRank of each supporting page is

$$\beta\frac{y}{m} + (1-\beta)/n$$

Where $\beta\frac{y}{m}$ represent the contribuion from t and $(1-\beta)/n$ is the supporting page's share of the $(1-\beta)$ of the PageRank that is distributed uniformly to all pages.

Now, we can compute the PageRank y of target page t. Its PageRank comes from three sources:

1. Contribution x from outside.
2. β times the PageRank of every supporting page:

$$\beta\left(\beta\frac{y}{m} + (1-\beta)/n\right)$$

3. $(1-\beta)/n$ is negligible and will be dropped to simplify the analysis.

From (1) and (2) we have:

$$
\begin{aligned}
y &= x + \beta m \left(\beta \frac{y}{m} + \frac{(1-\beta)}{n} \right) \\
&= x + \beta^2 y + \beta(1-\beta)\frac{m}{n} \\
&= \frac{x}{1-\beta^2} + \frac{\beta(1-\beta)}{1-\beta^2}\frac{m}{n} \qquad\qquad \text{solving for } y \\
&= \frac{x}{1-\beta^2} + c\frac{m}{n} \qquad\qquad\qquad \text{where } c = \frac{\beta}{(1+\beta)}
\end{aligned}
$$

3.6.3 Combating Link Spam

It has become essential for search engines to detect and eliminate link spam. The first approach is to identify structure such as that shown in Fig. 3.7 and eliminate it. This is not easy, because the spammers can disguise the structure of their spam farm. In the second approach a search engine can modify the PageRank to lower the rank of link spam pages.

- *TrustRank*: a variation of topic-sensisitve PageRank.
- *Spam mass*, a calculation that identifies the pages that are likely to be spamm and allow the search engine to lower their PageRank or eliminate them.

3.6.4 TrustRank

TrustRank is topic-sensitive PageRank, where the "topic" is a set of pages believed to be not spam. The theory is that while a spam page might easily be made to link to a trustworthy page, it is unlikely that a trustworthy page would link to a spam page.

To implement TrustRank, we need to develop a suitable teleport set of trustworthy pages. Two approaches that have been tried are:

1. Let humans examine a set of pages and decide which of them are trustworthy. For example, we might pick the pages of highest PageRank to examine and it is essentially impossible to give a spam page a PageRank near the top of the list.
2. Pick a domain whose membership is controlled, on th assumption that is hard for a spammer to get a page in that domain. For example, we could pick the .edu domain, since university pages are unlikely to be spam farms.

It is likely that search engines today implement a strategy of the second type routinely, so that what we think of as PageRank really is a form of TrustRank.

3.6.5 Spam Mass

The idea behind spam mass is that we measure for each page the fraction of its PageRank that come from spam. Suppose page p has PageRank r and the TrustRank t

based on some teleport set of trustworthy pages, then the *spam mass* of p is:

$$\text{SM}(p) = \frac{r - t}{r}$$

A negative or small positive smap mass means that p is probably not a spam page, while a spam mass close to 1 means that p is probably a spam page.

3.7 Hubs and Authorities

The algorithm for computing "hubs and authorities" is a variant of PageRank. This algorithm is sometimes called *HITS* (*Hyperlink-Induced Topic Search*) .

While PageRank assume a one-dimensional notion of importance for pages, HITS views important pages as having two different types of importance:

1. Pages that provide information about a topic. These pages are called *authorities*.
2. Pages that provide links to authorities, where to go to find out about that topic. These pages are called *hubs*.

HITS uses a mutually recursive definition of two concepts: "a page is a good hub if it links to good authorities, and a page is a good authority if it is linked to by good hubs."

3.7.1 Formalizing Hubbiness and Authority

We shall assign two score to each Web page. One score represents the *hubbiness* of a page and the other represents the *authority* of a page. Assuming that pages are enumerated, we represent these scores as two vectors **h** and **a**. In order to estimate the *hubbiness* and *authority* of a page, we can estimate the authority of successors and the hubbiness of predecessors, respectively. Normally scale the values of the vectors **h** and a so that the largest component is 1.

To describe the iteratie computation of **h** and **a**, we need to define the *adjacency/link matrix A* of the Web, $L_{n \times n}$ where n is the number of pages. The entry L_{ij} is 1 if there is a link from page i to page j and 0 otherwise. Let L^T be the transpose of L.
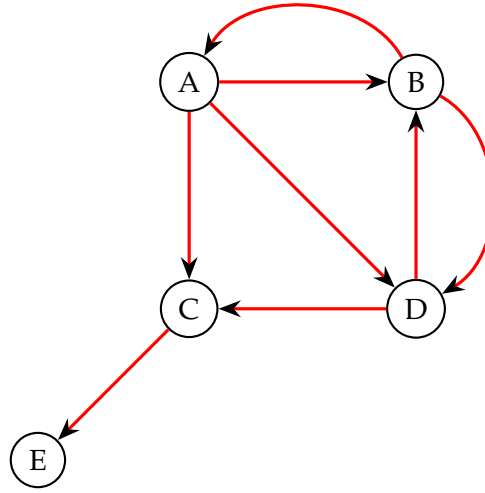
Figure 3.8. *Sample data for HITS.*

$$L = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad L^T = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Figure 3.9. *The link matrix L and its transpose L^T for the graph in Fig. 3.8.*

We can express:

$$\mathbf{h} = \lambda L \mathbf{a} \qquad \text{where } \lambda \text{ is the scaling factor}$$
$$\mathbf{a} = \mu L^T \mathbf{h} \qquad \text{where } \mu \text{ is the scaling factor}$$

And we can compute **h** and **a** independently:

$$\mathbf{h} = \lambda \mu L L^T \mathbf{h}$$
$$\mathbf{a} = \mu \lambda L^T L \mathbf{a}$$

However, since LL^T and $L^T L$ are not as sparse as L and L^T, we can use the following algorithm:

Algorithm 1: HITS algorithm - *Hyperlink-Induced Topic Search*

Input: A link matrix L

1 begin

2 $\mathbf{h} \leftarrow \mathbf{1}_n \equiv [1, 1, \ldots, 1]_n$

3   **while h** *and* **a** *have not converged* **do**

4 $\mathbf{h} \leftarrow L^T \mathbf{a}$

5 $\mathbf{a} \leftarrow L\mathbf{h}$

6   normalize **h** and **a**   `// largest component to 1`

7   **end**

8 end

Output: The hubbiness and authority vectors **h** and **a**

4
Finding Similar Items

We show how the problem of finding textually similar documents can be turned into such a set problem by the technique known as "shingling". 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. Another important problem that arises when we search for similar items of any kind is the problem of "nearest neighbor search". We show how this problem can be solved by a technique called "Locality Sensitive Hashing".

4.1 Application 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 is called *Jaccard similarity*:

$$\text{Jaccard}(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

where A and B are sets. The Jaccard similarity is a number between 0 and 1, where 0 means that the two sets are disjoint, and 1 means that they are equal.

4.1.1 Similarity of Documents

We can use the Jaccard similarity to measure the similarity of documents. We can represent a document as a set of words that occur in it. We can then measure the similarity of two documents by measuring the similarity of the sets of words that occur in them.

In the base case, where the two documents are exact duplicates, we can compare them character by character. However in many applications, the document are not identical, but they share a lot of part of text.

– Plagiarism: finding plagiarized documents tests our ability to find textually similar documents.

– Mirror Pages: a mirror page is a page that is identical to another page, except for some minor changes, such as the replacement of the author's name or the addition of a few words.

– Articles from the Same Source: we may want to find articles from the same source, such as the New York Times.

4.1.2 Collaborative Filtering as a Similar-Sets Problem

Another class of applications where similarity of sets is very important is called *collaborative filtering*, a process where we raccomend to users items that other similar users have liked.

On-line Purchases
We can use collaborative filtering to recommend products to users based on the products that other similar users have purchased. We can say that two users are similar if their sets of purchased products have a high Jaccard similarity.

Collaborative filtering requires us to find the nearest neighbors of a given user. However, by combining the similarity-finding with clustering, we are able to find the nearest neighbors of a user very quickly.

Movie Ratings
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.

When out data consists of rating rather then binary decisions, we cannot use the sets of items. Some alternative approaches are:

1. Ignore low-rated customer/movie pairs.
2. Use a threshold to convert ratings to binary decisions.
3. If ratins are on a scale of 1 to 5, we can put in a multi-set the number of times each movie was rated. In order to compute Jaccard similarity of a multi-set, we can use the following formula:

$$\text{Jaccard}(A, B) = \frac{\sum_{i=1}^{n} \min(A_i, B_i)}{\sum_{i=1}^{n} \max(A_i, B_i)}$$

4.2 Shingling of Documents

The most effective way to represent documents as sets is to construct from the document the set of short string of characters that appear within it. These strings are called *shingles*.

A k-shingle of a document is simply a sequence of k characters that appears in it. For example, suppose that D is the document containing the string "`abcdadb`" and that $k = 2$. Then the set of 2-shingles for D is $\{$`ab`, `bc`, `cd`, `da`, `db`$\}$.

4.2.1 Choosing the Shingle Size

We can pick k to be any constant we like. But, if k is too small, then the Jaccard similarity between two documents will be very high, even if the documents are completely different. On the other hand, if k is too large, then we will miss the similarity between documents that share a lot of common text.

The important thing to remember is that, k should be picked large enough that the probability of a given shingle appearning in any given document is low.

A good rule of thumb is to immagine that there are only 20 characters and estimate the number of k-shingles as 20^k. For example, for an email we can choose $k = 5$, since $20^5 = 3.2$ million and an email is unlikely to contain more than a few million characters. For a web page, we can choose $k = 9$, since $20^9 = 5.4$ trillion and a web page is unlikely to contain more than a few trillion characters.

4.2.2 Hashing Shingles

Instead of using substring directly as shingles, we can pick a hash function h and use $h(S)$ as the shingle for any string S. For instance, we can pick $k = 9$ and use the hash function $h(S) = \{0, 1, \ldots, 2^{32} - 1\}$, we need to store only 4 bytes for each shingle.

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.

4.2.3 Shingles Built from Words

An alternative form of shingle has proved effective for the problem of identifying similar news articles, as mentioned in Section 4.1.1. 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. 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.

4.3 Similarity-Preserving Summaries of Sets

Set of shingles are large. Even if we hash them to four bytes each, the set needed to store a set is still large. The goal of this section is to replace large sets by much smaller representation called "signatures" that preserve the similarity of the sets.

The important property we need for signature is that we can compare the signature of two sets and get a good estimate of their Jaccard similarity. It is not possible that the signature of two sets are identical but their Jaccard similarity is low.

4.3.1 Matrix Representation of Sets

Before explaining how it is possible create a small signature from a large set, we need to introduce a matrix representation of sets. The columns of the matrix correspond to the sets, and the row correspond to the elements of the universal sets from which elements of the sets are drawn. Given a row i and a column j, the entry M_{ij} is 1 if the element i is in the set j and 0 otherwise.

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 4.1. *Matrix representation of sets.*

It is important remember that the characteristic matrix is unlikely to be the way the data is stored in practice. The matrix representation is just a convenient way to think about sets.

4.3.2 Minhashing

The signature we deisre to construct for sets are composed of the results of a large number of calulation each of which is a "minhash" of the characteristic matrix.

To *minhash* a set represented by a column of the characteristic matrix, we first randomly permute the rows of the matrix. Then, the minhash value of the set is the number of the first row in which the set has a 1.

$$h(S) = \min_{i \in S} \pi(i)$$
$$\text{where } \pi(x) = \alpha x + \beta \bmod n$$
$$\text{and } \alpha, \beta \text{ are random integers,}$$
$$n \text{ is number of rows in the matrix}$$

4.3.3 Minhashing and Jaccard Similarity

The minhashing technique has the property that the probability that the minhash of two sets is the same is equal to the Jaccard similarity of the sets.

To see why, we need to picture the columns for those two sets. If we restrict our attention to the column S_1 and S_2, then rows can divided into three categories:

1. X rows have 1 in both S_1 and S_2.
2. Y rows have 1 in only one of S_1 and S_2 and 0 in the other.
3. Z rows have 0 in both S_1 and S_2.

Remembering that the matrix is sparse, most rows are in category 3 (Z). However, it is the ratio between X and Y that determines the Jaccard(S_1, S_2) and the $\mathbb{P}(h(S_1) = h(S_2))$.

Let $x \in X$ and $y \in Y$, then:

$$\text{Jaccard}(S_1, S_2) = x/(x+y)$$
$$\text{where } x = |S_1 \cap S_2|$$
$$x + y = |S_1 \cup S_2|$$

Consider the probability, immagine the rows permuted randomly and proceed from top. The $\mathbb{P}$(we meet a row in X before we meet a row in Y) $= x/(x+y)$. If the first row we meet is in X, then $h(S_1) = h(S_2)$, but if the first row we meet is in Y, then $h(S_1) \neq h(S_2)$. We can conclude that:

$$\mathbb{P}(h(S_1) = h(S_2)) = \mathbb{P}(\text{we meet a row in } X \text{ before we meet a row in } Y)$$
$$= x/(x+y)$$
$$= \text{Jaccard}(S_1, S_2)$$

4.3.4 Minhash Signature

Let M our characteristic matrix. To represent sets, we pick at random some number n of premutations of the rows of M. 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* vecor $[h_1(S), h_2(S), \ldots, h_n(S)]$ for S. Now, we can form the *signature matrix* $M\prime$, where the i-th column of M is replaced by the minhash signature vector. Note that the signature matrix $M\prime$ has the same number of columns as M but only n rows.

It is not feasible to permute randomly a large characteristic matrix. However, it is possible to simulate a random permutation by a random hash function that map row numbers to as many buckets as there are rows. A hash function that maps integers $0, 1, \ldots, n-1$ to bucket numbers $0, 1, \ldots, n-1$, typically will map some pair of integers to the same bucket and leave some buckets empty.

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

1. Compute $h_1(r), h_2(r), \ldots, h_n(r)$. For example, $h_1(r) = x + 1 \mod 5$ and $h_2(r) = 3x + 1 \mod 5$.

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 $\text{SIG}(i, c) = \min(\text{SIG}(i, c), h_i(r))$.

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 4.2. *Hash functions computed for the matrix of Figure 4.1..*

Notice that these two simple hash functions are true permutations of the tows, but a true permutations is only possible because the number of rows, 5, is prime.

Now, simulate the algorithm for computing the **signature matrix**. Initially, the matrix has only ∞ values.

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

Consider the first row of the matrix of

NowFigure 4.2. We see that the values of $h_1(0)$ and $h_2(0)$ are both 1. In the first row, only S_1 and S_4 are 1, so only this columns are updated comparing the values of $h_1(0)$ and $h_2(0)$ with the ∞ (current values). The result is the following:

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

Consider the second row of the matrix of Figure 4.2. This row has value 1 only in S_3, and $h_1(1) = 2$ and $h_2(1) = 4$. Thus, we set $\text{SIG}(1,3) = 2$ and $\text{SIG}(2,3) = 4$. The result is the following:

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

The row numbered 3 of Figure 4.2 has 1 in S_2 and S_4 and the values of $h_1(2) = 3$ and $h_2(2) = 2$. Thus, we set $\text{SIG}(1,2) = 3$ and $\text{SIG}(2,2) = 2$. We could also set $\text{SIG}(1,4) = 3$ and $\text{SIG}(2,4) = 2$, but we do not do this because we already have a smaller value for $\text{SIG}(1,4)$ and $\text{SIG}(2,4)$. The result is the following:

	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 Figure 4.2. This row has 1 in S_1, S_3 and S_4 and the values of $h_1(3) = 4$ and $h_2(3) = 0$. The value of $h_1(3)$ is greater than all values of the first row. However, the value 0 is smaller then the values that are already present in the second row. Thus, we set $\text{SIG}(2,1) = 0$ and $\text{SIG}(2,3) = 0$ and $\text{SIG}(2,4) = 0$. The result is the following:

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

Finally, consider the row of Figure 4.2 numbered 4. This row has 1 only in S_3 and the values of $h_1(4) = 0$ and $h_2(4) = 3$. Thus, we compare $[2, 0]$ with $[0, 3]$ and set SIG$(1, 3) = 0$ and keep SIG$(2, 3) = 0$ because it is already smaller. The result is the following:

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

We can estimate the Jaccard similarities of the underlying sets from this signature matrix. Column S_1 and S_4 are identical, so Jaccard$(S_1, S_4) = 1$ but if we look at Figure 4.2, we see that the Jaccard similarity of S_1 and S_4 is 2/3. This is because the signature matrix is only an approximation of the Jaccard similarities, and this example is to small for the law of the large numbers to ensure that the estimate is close.

4.4 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.

We need to focus our attention only on pairs that are likely to be similar, without investigating all pairs. This is the idea behind **locality-sensitive hashing** (LSH).

4.4.1 LSH for Minhash Signatures

One general approach to LSH it to "hash" items several times, doing it in such a way that if two items are similar, then the probability that they will collide is high. We can consider any pair that hashed to the same bucket as a *candidate pair*.

We hope that dissimilar pairs will not collide. Notice that this is not guaranteed and we call *false positives* the pairs that are dissimilar but collide. On the other hand, we call *false negatives* the pairs that are similar but do not collide, and we hope that their number is small.

If we minhash signature for the items, an effective way to choose the hashing is to divide the signature matrix into b bands of r rows each. For each band, we hash the r rows into a hash table with n (big enough) buckets. We can use the same hash function for all bands but we use a different bucket array for each band, so column with the same vector in different buckets will not in the same bucket.

		1 0 0 0 2	
Band 1	...	3 2 1 2 2	...
		0 1 3 1 1	
Band 2			
Band 3			
Band 4			

Figure 4.3. *Dividing a signature matrix into four bands of three rows per band.*

Figure 4.3 show part of a signature matrix of 12 rows divided into four bands of three rows each. The second and fourth columns are identical, so they will be hashed to the same bucket in the first band and the columns that are different will be hashed to different buckets. Two columns that do not agree in Band 1 have three other chances to become a candidate pair. However, if two columns are similar there is a good chance that they will be hashed to the same bucket in at least one of the bands.

4.4.2 Analysis of the Banding Technique

Suppose we use b bands of r rows, and suppose that a particular pair of documents have Jaccard similarity s. Recall from Section 4.3.3 that the probability that two columns agree in a particular row is s. We can calculate the probability that two signature become a candidate pair as follows:

$$\mathbb{P}(\text{agree in all row of one band}) = s^r$$
$$\mathbb{P}(\text{disagree in at least one row of one band}) = 1 - s^r$$
$$\mathbb{P}(\text{disagree in at least one row of all bands}) = (1 - s^r)^b$$
$$\mathbb{P}(\text{agree in all the rows of a band and become a candidate pair}) = 1 - (1 - s^r)^b$$

After choosing r and b, this function has the form of an *S-curve* as shown in Figure 4.4. Where in the *x*-axis we have the Jaccard similarity s and in the *y*-axis we have the probability that two signatures become a candidate pair.
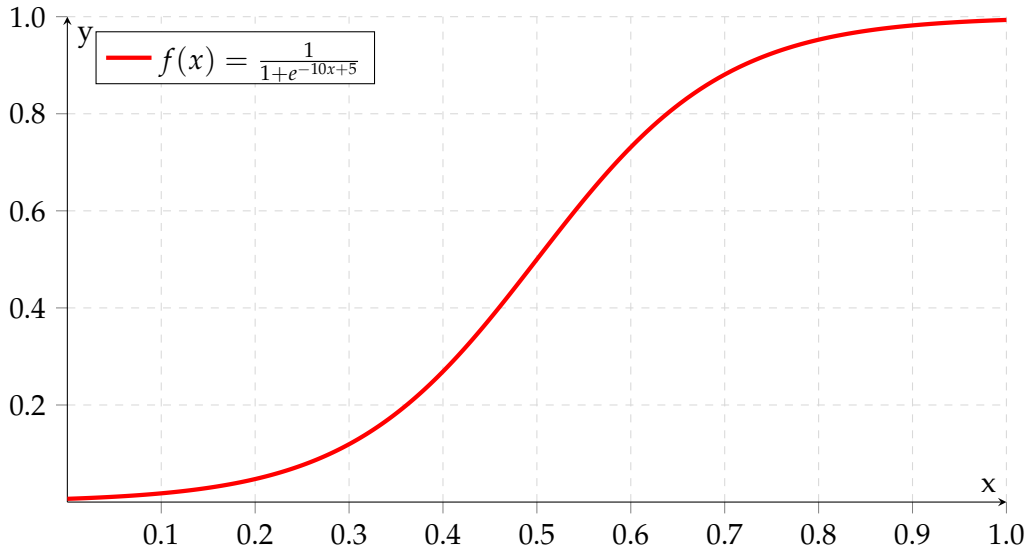
Figure 4.4. *The S-curve.*

The *threshold* of similarity s at which the probability of becoming a candidate pair is $1/2$ is a function of b and r. If b and r are large, the threshold is high, so only pairs of documents that are very similar will be candidates. If b and r are small, the threshold is low, so many pairs of documents will be candidates. An approximation of the threshold is $1/b^{1/r}$.

4.4.3 Combining the Techniques

We now give an approach to finding the set of candidate pairs for similar documents and then discovering the truly similar documents among the candidates. This approach can produce *false positives* and *false negatives*.

1. Pick a value 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 pair to orger them by shingle.
3. Pick a length n for the minhash signatures. Feed the sorted list to the algorithm of Section 4.3.4 to produce the minhash signature for each document.
4. Choose a threshold t for Jaccard similarity. Pick a number of bands b and a number of rows r per band, such that $br = n$. If the presence of *false negatives* is not acceptable, then choose b and r to produce a threshold lower than t.
5. Construct candidate pairs by applying the LSH technique of Section 4.4.1.
6. Examine each candidate pair's signatures and determined if the fraction of components in which they agree is at least t.
7. Optionally, if the signature are sufficiently similar, go to the documents themselves and check that they are truly similar.

4.5 Distance Measures

The Jaccard similarity is a measure of how close sets are, it is not really a distance measure. Jaccard distance is defined as $1 -$ Jaccard similarity and it is a distance measure, but there it is not the only one.

In order to define a distance measure, we need to define a *distance function $d(x, y)$* that satisfies the following axioms:

1. $d(x, y) \geq 0$ for all x and y. (no negative distances)
2. $d(x, y) = 0$ iif $x = y$. (distances are positive, except for identical points)
3. $d(x, y) = d(y, x)$. (distance is simmetric)
4. $d(x, z) \leq d(x, y) + d(y, z)$. (triangle inequality)

4.5.1 Euclidean Distance

An *n-dimensional Euclidean space* is one where points are vectors of n real numbers. The Euclidean distance between two points x and y in an n-dimensional Euclidean space, which we shall refer to as the *L_2-norm*, is defined as:

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

It is easy to verify the axioms of a distance function for the Euclidean distance. The Euclidean distance cannot be negative because it is the square root of a sum of squares, and the sum of square are strictly positive. On the other hand, the distance will be 0 if $x_i = y_i$ for all i, which means that $x = y$. Simmetry follows because $(x_i - y_i)^2 = -(y_i - x_i)^2$. Finally, the triangle inequality follows from the fact that the square root is a concave function, so the sum of square roots is less than the square root of the sum.

There are other distance measures that have been used for Euclidean spaces. For any constant r, we can define the *L_r-norm* as:

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

The L_1-norm is also known as the *Manhattan distance* or *city-block distance*. It is the distance between two points in a city where you can only travel along the streets. The

L_∞-norm is the limit as r approaches infinity. As r gets larger, only the dimension with the largest difference between x and y matters. Thus, the L_∞-norm is the maximum of $|x_i - y_i|$ over all i.

4.5.2 Jaccard Distance

The *Jaccard distance* of sets x and y is defined as $d(x,y) = 1 - \text{Jaccard}(x,y)$. Recall that the Jaccard similarity of two sets is define as follows:

$$\text{Jaccard}(x,y) = \frac{|x \cap y|}{|x \cup y|}$$

We have to verify that this is a distance function:

1. $d(x,y)$ is non-negative because the size of the intersection cannot exceed the size of the union.
2. $d(x,y) = 0$ if and only if $x = y$ because $x \cap x = x \cup x = x$.
3. $d(x,y) = d(y,x)$ because the union and intersection are symmetric.
4. For the triangle inequality, recall from Section 4.3.3 that $\text{Jaccard}(x,y)$ is the probability that a random minhash function maps x and y to the same value. 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 $h(x) = h(y)$ is at least as large as the probability that $h(x) = h(z)$ plus the probability that $h(z) = h(y)$. This is true because the probability that $h(x) = h(y)$ is at least as large as the probability that $h(x) = h(z)$ plus the probability that $h(z) = h(y)$.

4.5.3 Cosine Similarity

The *cosine similarity* makes sense in spaces that have dimensions, including Euclidean spaces and discrete version of Euclidean spaces, such as spaces where points are vectors with ingeger components of Boolean (0 and 1) components. The cosine distance between two points is the angle that the vectors to those points make with the origin.

We can calculate the cosine distance by first calculating 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 $x \cdot y$ divided by the L_2-norms of x and y:

$$\text{cosine}(x,y) = \frac{x \cdot y}{\|x\|\|y\|}$$
$$\text{where } x = [x_1, x_2, \ldots, x_n] \text{ and}$$
$$y = [y_1, y_2, \ldots, y_n] \text{ and}$$
$$x \cdot y = \sum_{i=1}^{n} x_i y_i$$

4.5.4 Edit Distance

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

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 a subsequence of both x and y.

The edit distance $d(x, y)$ can be calculated as the length of x plus the length of y minus twice the length of an LCS of x and y.

4.5.5 Hamming Distance

Given a space of vectors, we define the *Hamming Distance* between two vecotrs to be the number of components in which they differ. Clearly, the Hamming distance cannot be negative, and it is zero if and only if the two vectors are identical. The triangle inequelity holds because if x and y differ in k components, and y and z differ in l components, then x and z differ in at most $k + l$ components.

Hamming distance is used when the vector are Boolean.

4.6 The Theory of Locality-Sensitive Functions

The LSH technique developed in Section 4.4 is one example of a family of functions (the minhash functions) that can be combined (by the binding technique) to distinguish strongly between pairs at a low distance from pairs at a high distance.

We shall explore other families of functions that can be used to produce a candidate pair efficiently. These functions can apply to the space of sets and Jaccard distance, or another space and distance function.

There are three conditions that we need for a family of functions:

1. They must be more likely to make close pairs be candidate pairs than distant pairs.

2. They must be statistically independent, it is possible to estimate the probability that functions will all give the same answer by the product rule for independent events.

3. They must be easy to compute.

    (a) They must be able to identify candidate pairs in time much less than the time take to look at all pairs. For example, minhashing is linear in the number of elements in the set.

    (b) They must be combinable to build functions that are better at avoiding false negatives and false positives, and the combined functions must be easy to compute.

4.6.1 Locality-Sensitive Functions

We consider functions that take two items and render a decision about whether these items should be a candidate pair.

In many cases, the function f is a hash function and the decision is based on whether or not the result is equal. We use $f(x) = f(y)$ to denote that $f(x, y) = "yes"$. We call *family* of functions a collection of functions in this form. For example, the family of minhash functions, each based on one of the possible permutations of rows of a characteristic matrix, form a family.

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

  – If $d(x, y) \leq d_1$, then $\mathbb{P}(f(x) = f(y)) \geq p_1$.
  – If $d(x, y) \geq d_2$, then $\mathbb{P}(f(x) = f(y)) \leq p_2$.

We have only one way to find a family of locality-sensitive functions: use the faimly minhash functions, and assume that the distance is calculated by the Jaccard distance.

4.6.2 Amplifying a Local-Sensitive Family

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

Since the members of $\mathcal{F}$ are independently chosen to make a mamber of $\mathcal{F}'$, we can assert that $\mathcal{F}'$ is a $(d_1, d_2, (p_1)^r, (p_2)^r)$-sensitive family.

There is another construction, which we call the *OR-construction*, that turns a (d_1, d_2, p_1, p_2)-sensitive family $\mathcal{F}$ into a $(d_1, d_2, 1 - (1 - p_1)^b, 1 - (1 - p_2)^b)$-sensitive family $\mathcal{F}'$. We define $f(x) = f(y)$ if and only if $f_i(x) = f_i(y)$ for at least one $i = 1, 2, \ldots, b$.

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

We can combine the AND-construction and the OR-construction in any order to make the lower probability close to 0 and the higher probability close to 1.

For example, suppose we start a family $\mathcal{F}$. We use the AND-construction with $r = 4$ to produce a family $\mathcal{F}'$. We then apply the OR-construction to $\mathcal{F}'$ with $b = 4$ to produce a third family $\mathcal{F}''$.

It means that $\mathcal{F}'$ is a quadruple of functions from $\mathcal{F}$, and $\mathcal{F}''$ is a quadruple of quadruples of functions from $\mathcal{F}$. Thus, $\mathcal{F}''$ is a family of 16 functions from $\mathcal{F}$.

$$\mathcal{F} = (0.2, 0.6, 0.8, 0.1) - sensitive$$
$$\mathcal{F}'' = (0.2, 0.6, 1 - (1 - 0.8)^4, 1 - (1 - 0.1)^4) - sensitive$$
$$= (0.2, 0.6, 0.8785, 0.0985) - sensitive$$

4.6.3 LSH Families for Hamming Distance

Suppose we have a space of d-dimensional vectors, and $h(x, y)$ denotes the Hamming distance between x and y. We can define $f_i(x)$ to be the i-th bit of x. Then, $f_i(x) = f_i(y)$ if and only if x and y agree in the i-th bit. The probability that $f_i(x) = f_i(y)$ is $1 - h(x, y)/d$. This situation is almost like the one we encountered for minhashing.

Thus, the family $\mathcal{F}$ of functions $\{f_1, f_2, \ldots, f_d\}$ is $(d_1, d_2, 1 - p_1/d, 1 - p_2/d)$-sensitive faimly of hash functions, for any $d_1 < d_2$.

This family is composed by d functions and this is limiting to apply AND-construction or OR-construction.

4.6.4 Random Hyperplanes and the Cosine Distance

Recall from Section 4.5.3 that the cosine distance between two vectors is the angle between the two vectors.
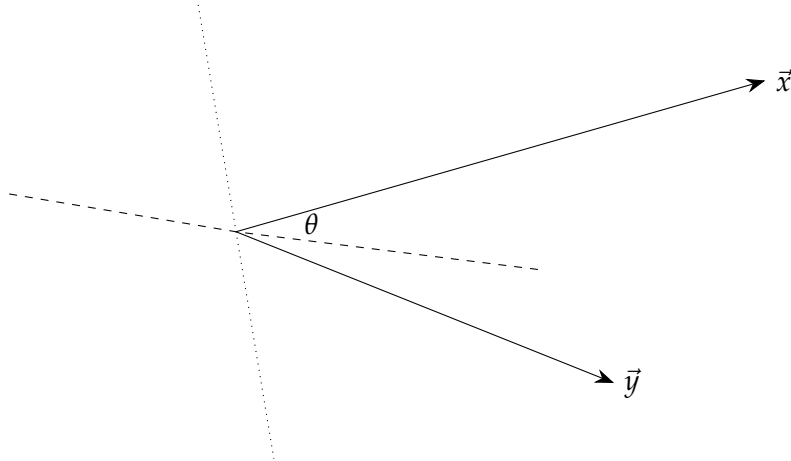

Figure 4.5. *Two vectors make an angle θ.*

Suppose we pick a hyperplane through the origin. The hyperplane divides the space into two half-spaces. First condider a vector v that is the dashed line in Figure 4.5, note that x and y are on different sides of the hyperplane. Then the dot product $v \cdot x$ and $v \cdot y$ have opposite signs.

On the other hand, the randomly chosen vector v could be normal to a hyperplane like the dotted line in Figure 4.5. In this case, x and y are on the same side of the hyperplane. Then the dot product $v \cdot x$ and $v \cdot y$ have the same sign.

The probability that $v \cdot x$ and $v \cdot y$ have the same sign is $\theta/180$. Thus, each hash function f in our locality-sensitive family $\mathcal{F}$ is defined by a randomly chosen vector v_f. Given two vectors x and y, then $f(x) = f(y)$ if and only if $v_f \cdot x$ and $v_f \cdot y$ have the same sign. The parameters are essentially the same as for the Jaccard-distance family. That is, $\mathcal{F}$ is a $(d_1, d_2, (180 - d_1)/180, (180 - d_2)/180)$-sensitive family.

4.6.5 LSH Families for Euclidean Distance

We will start with a 2-dimensional Euclidean space. Each hash function f in our family $\mathcal{F}$ will be associated with a randomly chosen line in this space. Pick a constant a and divide the line into segments of length a, as shown in Figure 4.6.

The segment of the line are the buckets into which function f hashed the points. A point is hashed to the bucket in which its projection onto the line lies. If the distance d between two points is small compared to a, then there is a good chance that the two points will be hashed to the same bucket. If d is large compared to a, then there is a small chance that the two points will be hashed to the same bucket.

However, suppose d is larger the a. In order for there to be any chance of two points falling in the same bucket, we need $d \cos \theta < a$, where θ is the angle between the line and the line connecting the two points.
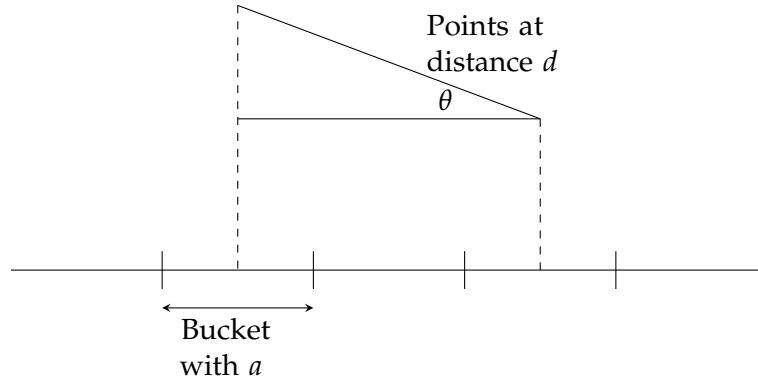

Figure 4.6. *Two points at distance $d \gg a$ have a small chance of being hashed to the same bucket.*

If $d \geq 2a$, then there is no more than a 1/3 chance the two points fall in the same bucket. The reason is that for $\cos \theta$ to be less than 1/2, we need to have $\theta > 60°$ and $\theta < 90°$. If $\theta > 0°$ and $\theta < 60°$, then $\cos \theta > 1/2$. But since θ is the smaller angle between two randomly chosen lines in the plane, θ is twice as likely to be between $0°$ and $60°$ as it is to be between $60°$ and $90°$.

We conclude that the family $\mathcal{F}$ just described froms a $(a/2, 2a, 1/2, 1/3)$-sensitive family of hash functions. That is, for istances up to $a/2$ the probability that two points are hashed to the same bucket is at least 1/2, while for distances greater than $2a$ the probability is at most 1/3.

5

Frequent Itemsets

The problem of finding frequent itemsets differs from the similarity search discussed in Chapter 4. Here we are interested in the absolute number of baskets that contain a particular set of items. In Chapter 4 we wanted items that have a large fraction of their baskets in common, even if that fraction is small.

5.1 The Market-Basket Model

The *market-basket* model of data is used to describe a common form of many-many relationship between two kinds of objects. We have *items* and *baskets*, sometimes called "transaction". Each basket consists of a set of items (an *itemset*), and usually we assume that the number of items in a basket is small compared to the total number of items. The number of baskets is usually assumed to be large, they cannot all be stored in memory.

5.1.1 Definitions of Frequent Itemsets

A set of items that appears in many baskets is said "frequent". If I is a set of items, the *support* for I is the number of baskets for which I is a subset. We say I is *frequent* if its support is greater than some *support threshold s*.

Each set is a basket, and the word are items. Ignore capitalization.

$S_1 = \{\text{Cat, and, Dog, bites}\}$

$S_2 = \{\text{Yahoo, news, claims, a, cat, mated, with, a, dog, and, produced, viable, offspring}\}$

$S_3 = \{\text{Cat, killer, likely, is, a, big, dog}\}$

$S_4 = \{\text{Professional, free, advice, on, dog, training, puppy, training}\}$

$S_5 = \{\text{Cat, and, kitten, training, and, behavior}\}$

$S_6 = \{\text{Dog, \&, Cat, provides, dog, training, in, Eugene, Oregon}\}$

$S_7 = \{\text{"Dog, and, cat", is, a, slang, term, used, by, police, officers, for, a, male-female, relationship}\}$

$S_8 = \{\text{Shop, for, your, show, dog, grooming, and, pet, supplies}\}$

Since the empty set is a subset of any set, the support for $\emptyset$ is 8.

Among the singleton sets, {cat} and {dog} are frequent. "Dog" appears in all but basket S_5, so its support is 7. "Cat" appears in all but S_4 and S_8, so its support is 6 etc.

Suppose that we set our threshold at $s = 3$. Then there are five frequent singleton items {cat}, {dog}, {and}, {a}, and {training}.

	training	a	and	cat
dog	4, 6	2, 3, 7	1, 2, 7, 8	1, 2, 3, 6, 7
cat	5, 6	2, 3, 7	1, 2, 5, 7	
and	5	2, 7		
a	none			

Figure 5.1. *Occurences of doubletons.*

Now, let us look at the doubletons. A doubleton cannot be frequent unless both items in the set are frequent by themselves. There are only ten possibile frequent doubletons. Figure 5.1 is a table indicating which baskets contain which doubletons.

There are only five frequent doubletons if $s = 3$, they are the ones that have at least three baskets in common. We can do for triple, etc.

5.1.2 Application of Frequent Itemsets

The original application of the market-basket model was in the analysis of true market baskets. For example, by finding frequent itemsets, a retailer can learn what is commonly purchased together, and use that information to decide how to arrange items in the store.

However, applications of frequent-itemsets analysis is not limited to market baskets. The same model can be used to mine many other kinds of data. Some examples are:

1. *Related concepts*: Let items be words, and let baskets be documents. A basket/document contains those items/words that are present in the document. However, if we ignore all the most common words, then we would hope to find among the frequent pairs some pairs of words that represent a joint concept.

2. *Plagiarism*: Let the items be documents and the baskets be sentences An item/document is "in" a basket/sentence if the sentence is in the document. In this application, we look for pairs of items that appear together in several baskets.

5.1.3 Association Rules

While the subject of this chapter is extracting frequent sets of items from data, this information is often presented as a collection of if–then rules, called *assciation rules*. The form of an association rule is $I \rightarrow j$, where I is a set and j is an item.

This implication say that if all items in I appear in some busket, then j is likely to appear to. With likely meaning the uspport for $I \cup \{j\}$ (the *confidence*), the basket that contains all items in I and j.

We define the *interest* of an association rule $I \rightarrow j$ to be the difference between the confidence of the rule and the support of j alone.

5.1.4 Finding Association Rules with High Confidence

Identifying useful association rules is not much harder than finding frequent itemsets. If we are looking for association rules $I \rightarrow j$ that apply to a reasonable number of baskets, then the support for I must be reasonably high.

Suppose we have found all itemsets that are smaller than our threshold, and we have exact support calculated for each of these itemsets. We can find within them all the association rules that have both high support and high confidence.

Let J be the set of n items, then there are only n possibile association rules of the form $I - j \rightarrow j \ \forall j \in J$. If J is frequent, $J - j$ is frequent for all $j \in J$ and it is frequent itemset, and we have already calculated its support. Their ratio is the confidence of the rule $J - j \rightarrow j$.

5.2 Market Baskets and the A-Priori Algorithm

The original improvement on the obvious algorithms, known as "A-Priori", from which many variants have been developed, will be covered here.

5.2.1 Representation of Market Basket Data

We assume that market-basket data is stored in a file basket-by-basket. Possibly, the data is in a distributed file system as in chapter 2.

We could imagine that such a file begins:

$$\{23, 456, 1001\}\{3, 18, 92, 145\}\{ \dots$$

The characters $\{\}$ are used to delimit baskets. And the items in a basket are separated by commas. Thus, the first basket contains items 23, 456, and 1001. The second basket contains items 3, 18, 92, and 145, etc.

It may be that one machine receives the entire file, or we could be using MapReduce or a similar tool to divide the work among many processors.

We also assume that the size of the file of baskets is sufficiently large that it cannot be stored in main memory. Thus, a major cost of any algorithm is the time it take to read the file. One a disk block full of baskets is in main memory, we can expand it, generaring all the subsets of size k.

If there are 20 items in a basket, then there are $\binom{20}{2} = 190$ pairs of items in the basket, and these can be generated easily.

As the size of subsets we want to generate gets larger, the number of subsets grows exponentially. In fact takes approximately n^k/k to genreate all the subsets of size k for basket with n items. Usually, we use $k = 2$ or 3.

Moreover, all the algorithms we discuss have the property that they read the basket file sequentially. Thus, algorithms can be characterized by the number of passes they make over the file, and we can controll only this parameter.

5.2.2 Use of Main Memory for Itemsets Counting

All frequent-itemset algorithms require us to maintain many different counts as we make a pass through the data. For example, we might need to count the number of items that each pair of items occurs in a baskets. If we do not have enough main memory to store all these counts, so this is a limitation.

It is not trivial to store the $\binom{n}{k}$ counts for all the k-itemsets. We can use a hash table, but we need to be careful to avoid collisions.

Bibliography