12.1.3 Serial Algorithms

Various algorithms have been developed to find frequent itemsets and association rules. The most famous one for the former task is the **Apriori** algorithm. Even it has many forms. We will discuss one of the simplest forms here.

The algorithm is basically a breadth-first tree search. At the root we find the frequent 1-item itemsets. In the online bookstore, for instance, this would mean finding all individual books that appear in at least r of our sales transaction records, where r is our threshold.

At the second level, we find the frequent 2-item itemsets, e.g. all pairs of books that appear in at least r sales records, and so on. After we finish with level i, we then generate new candidate itemsets of size i+1 from the frequent itemsets we found of size i.

The key point in the latter operation is that if an itemset is not frequent, i.e. has support less than the threshold, then adding further items to it will make it even less frequent. That itemset is then pruned from the tree, and the branch ends.

Here is the pseudocode:

```
set F_1 to the set of 1-item itemsets whose support exceeds the threshold
for i = 2 to b
    F_i = φ
    for each I in F_{i-1}
        for each K in F_1
            Q = I ∪ K
            if support(Q) exceeds support threshold
              add Q to F_i
    if F_i is empty break
return ∪_i F_i
```

Again, there are many refinements of this, which shave off work to be done and thus increase speed. For example, we should avoid checking the same itemsets twice, e.g. first $\{1,2\}$ then $\{2,1\}$. This can be accomplished by keeping itemsets in lexicographical order. We will not pursue any refinements here.

12.1.4 Parallelizing the Apriori Algorithm

Clearly there is lots of opportunity for parallelizing the serial algorithm above. Both of the inner **for** loops can be parallelized in straightforward ways; they are "embarrassingly parallel." There are of course critical sections to worry about in the shared-memory setting, and in the message-passing setting one must designate a manager node in which to store the F_i.

However, as more and more refinements are made in the serial algorithm, then the parallelism in this algorithm become less and less "embarrassing." And things become more challenging if the storage needs of the F_i, and of their associated "accounting materials" such as a directory showing the current tree structure (done via hash trees), become greater than what can be stored in the memory of one node.

In other words, parallelizing the market basket problem can be very challenging. The interested reader is referred to the considerable literature which has developed on this topic.

12.2 Probability Density Estimation

Let X denote some quantity of interest in a given population, say people's heights. Technically, the **probability density function** of X, typically denoted by f, is a function on the real line with the following properties:

- $f(t) \geq 0$ for all t
- for any $r < s$,

$$P(r < X < s) = \int_r^s f(t)\, dt \tag{12.1}$$

(Note that this implies that f integrates to 1.)

This seems abstract, but it's really very simple: Say we have data on X, n sample values $X_1, ..., X_n$, and we plot a histogram from this data. Then *f is what the histogram is estimating*. If we have more and more data, the histogram gets closer and closer to the true f.[2]

So, how do we estimate f, and how do we use parallel computing to reduce the time needed?

12.2.1 Kernel-Based Density Estimation

Histogram computation breaks the real down into intervals, and then counts how many X_i fall into each interval. This is fine as a crude method, but one can do better.

No matter what the interval width is, the histogram will consist of a bunch of rectanges, rather than a smooth curve. This problem basically stems from a lack of weighting on the data.

For example, suppose we are estimating f(25.8), and suppose our histogram interval is [24.0,26.0], with 54 points falling into that interval. Intuitively, we can do better if we give the points closer to 25.8 more weight.

[2]The histogram must be scaled to have total area 1. Most statistical programs have options for this.

One way to do this is called **kernel-based** density estimation, which for instance in R is handled by the function **density()**.

We need a set of weights, more precisely a weight function k, called the **kernel**. Any nonnegative function which integrates to 1—i.e. a density function in its own right—will work. Typically k is taken to be the Gaussian or normal density function,

$$k(u) = \frac{1}{\sqrt{2\pi}} e^{-0.5u^2} \tag{12.2}$$

Our estimator is then

$$\widehat{f}(t) = \frac{1}{nh} \sum_{i=1}^{n} k\left(\frac{t - X_i}{h}\right) \tag{12.3}$$

In statistics, it is customary to use the $\widehat{}$symbol (pronounced "hat") to mean "estimate of." Here $\widehat{f}$ means the estimate of f.

Note carefully that we are estimating an entire function! There are infinitely many possible values of t, thus infinitely many values of f(t) to be estimated. This is reflected in (12.3), as $\widehat{f}(t)$ does indeed give a (potentially) different value for each t.

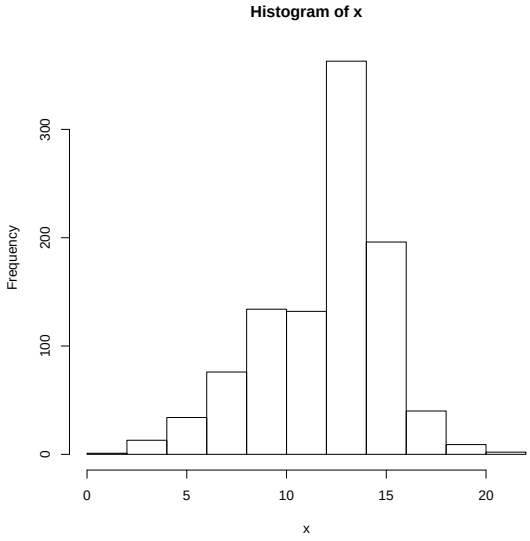
Here h, called the *bandwidth*, is playing a role analogous to the interval width in the case of histograms.

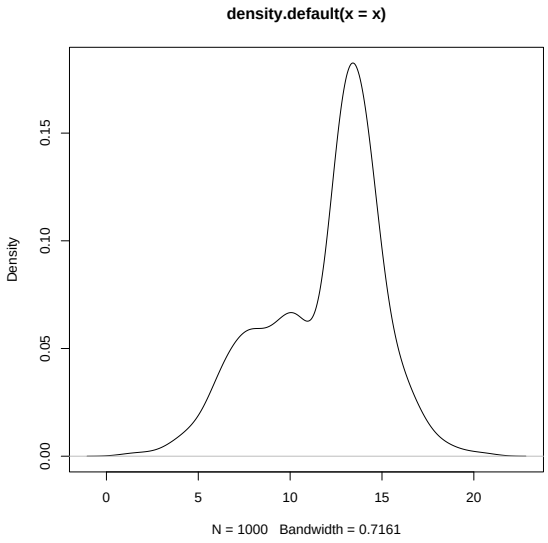
Again, this looks very abstract, but all it is doing is assigning weights to the data. Consider our example above in which we wish to estimate f(25.8), i.e. t = 25.8 and h = 6.0. If say, X_{88} is 1209.1, very far as awa from 25.8, we don't want this data point to have much weight in our estimation of f(25.8). Well, it won't have much weight at all, because the quantity

$$u = \frac{25.8 - 88}{6} \tag{12.4}$$

will be very large, and (12.2) will be tiny, as u will be way, way out in the left tail.

Now, keep all this in perspective. In the end, we will be plotting a curve, *just like we do with a histogram*. We simply have a more sophiticated way to do this than plotting a histogram. Following are the graphs generated first by the histogram method, then by the kernel method, on the same data:





There are many ways to parallelize this computation, such as:

- Remember, we are going to compute (12.3) for many values of t. So, we can just have each process compute a block of those values.

- We may wish to try several different values of h, just as we might try several different interval widths for a histogram. We could have each process compute using its own values of h.

- It can be shown that (12.3) has the form of something called a **convolution**. The theory of convolution

would take us too far afield,[3] but this fact is useful here, as the Fourier transform of a convolution can be shown to be the product of the Fourier transforms of the two convolved components.[4] In other words, *this reduces the problem to that of parallelizing Fourier transforms*—something we know how to do, from Chapter 11.

12.2.2 Histogram Computation for Images

In image processing, histograms are use to find tallies of how many pixels there are of each intensity. (Note that there is thus no interval width issue, as there is a separate "interval" value for each possible intensity level.) The serial pseudocode is:

```
for i = 1,...,numintenslevels:
   count = 0
   for row = 1,...,numrows:
      for col = 1,...,numcols:
         if image[i][j] == i: count++
   hist[i] = count
```

On the surface, this is certainly an "embarrassingly parallel" problem. In OpenMP, for instance, we might have each thread handle a block of rows of the image, i.e. parallelize the **for row** loop. In CUDA, we might have each thread handle an individual pixel, thus parallelizing the nested **for row/col** loops.

However, to make this go fast is a challenge, say in CUDA, due to issues of what to store in shared memory, when to swap it out, etc. A very nice account of fine-tuning this computation in CUDA is given in *Histogram Calculation in CUDA*, by Victor Podlozhnyuk of NVIDIA, 2007 `http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/histogram256/doc/histogram.pdf`.

---

[3] If you've seen the term before and are curious as to how this is a convolution, read on:
Write (12.3) as

$$\widehat{f}(t) = \sum_{i=1}^{n} \frac{1}{h} k\left(\frac{t - X_i}{h}\right) \cdot \frac{1}{n} \tag{12.5}$$

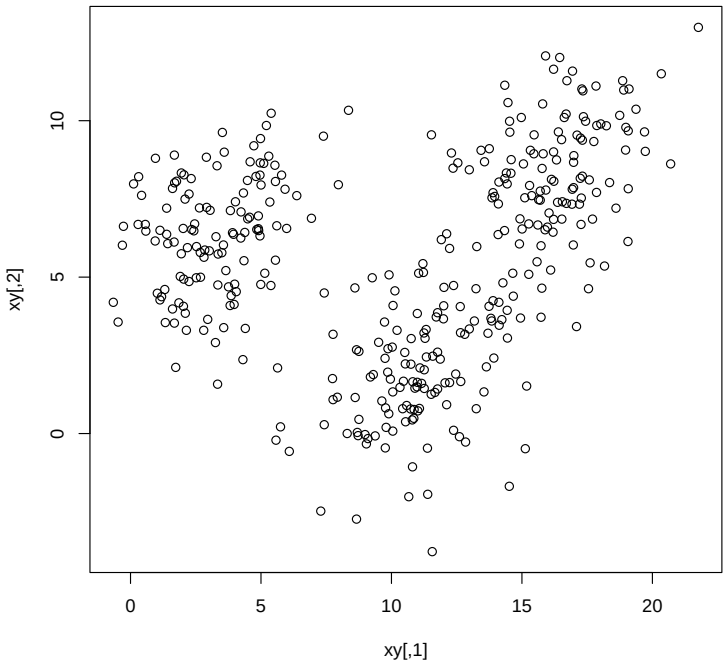
Now consider two artificial random variables U and V, created just for the purpose of facilitating computation, defined as follows. The random variable U takes on the values ih with probability $g \cdot \frac{1}{h} k(i)$, i = -c,-c+1,...,0,1,...,c for some value of c that we choose to cover most of the area under k, with g chose so that the probabilities sum to 1. The random variable V takes on the values $X_1, ..., X_n$ (considered fixed here), with probability 1/n each. U and V are set to be independent.

Then (g times) (12.5) becomes P(U+V=t), exactly what convolution is about, the probability mass function (or density, in the continuous case) of a random variable arising as the sum of two independent nonnegative random variables.

[4] Again, if you have some background in probability and have see characteristic functions, this fact comes from the fact that the characteristic function of the sum of two independent random variables is equal to the product of the characteristic functions of the two variables.

12.3 Clustering

Suppose you have data consisting of (X,Y) pairs, which when plotted look like this:



It looks like there may be two or three groups here. What clustering algorithms do is to form groups, both their number and their membership, i.e. which data points belong to which groups. (Note carefully that *there is no "correct" answer here. This is merely an exploratory data analysis tool.*

Clustering is used is many diverse fields. For instance, it is used in image processing for segmentation and edge detection.

Here we have to two variables, say people's heights and weights. In general we have many variables, say p of them, so whatever clustering we find will be in p-dimensional space. No, we can't picture it very easily of p is larger than (or even equal to) 3, but we can at least identify membership, i.e. John and Mary are in group 1, Jenny is in group 2, etc. We may derive some insight from this.

There are many, many types of clustering algorithms. Here we will discuss the famous **k-means** algorithm, developed by Prof. Jim MacQueen of the UCLA business school.

The method couldn't be simpler. Choose k, the number of groups you want to form, and then run this:

```
# form initial groups from the first k data points (or choose randomly)
for i = 1,...,k:
   group[i] = (x[i],y[i])
   center[i] = (x[i],y[i])
do:
   for j = 1,...,n:
      find the closest center[i] to (x[j],y[j])
      cl[j] = the i you got in the previous line
   for i = 1,...,k:
      group[i] = all (x[j],y[j]) such that cl[j] = i
      center[i] = average of all (x,y) in group[i]
until group memberships do not change from one iteration to the next
```

Definitions of terms:

- *Closest* means in p-dimensional space, with the usual Euclidean distance: The distance from $(a_1, ..., a_p$ to $(b_1, ..., b_p$ is

$$\sqrt{(b_1 - a_1)^2 + ... + (b_p - a_p)^2} \tag{12.6}$$

- The *center* of a group is its **centroid**, which is a fancy name for taking the average value in each component of the data points in the group. If p = 2, for example, the center consists of the point whose X coordinate is the average X value among members of the group, and whose Y coordinate is the average Y value in the group.

12.4 Principal Component Analysis (PCA)

Consider data consisting of (X,Y) pairs as we saw in Section 12.3. Suppose X and Y are highly correlated with each other. Then for some constants c and d,

$$Y \approx c + dX \tag{12.7}$$

Then in a sense there is really just one random variable here, as the second is nearly equal to some linear combination of the first. The second provides us with almost no new information, once we have the first. In other words, even though the vector (X,Y) roams in *two*-dimensional space, it usually sticks close to a *one*-dimensional object, namely the line (12.7).

Now think again of p variables. It may be the case that there exist r $<$ p variables, consisting of linear combinations of the p variables, that carry most of the information of the full set of p variables. If r is much less than p, we would prefer to work with those r variables. In data mining, this is called **dimension reduction**.

It can be shown that we can find these r variables by finding the r eigenvectors corresponding to the r largest eigenvalues of a certain matrix. We will not pursue that here, but the point is that again we have a matrix formulation, and thus parallelizing the problem can be done easily by using methods for parallel matrix operations.

Appendix A

Review of Matrix Algebra

This book assumes the reader has had a course in, or has self-studied, linear algebra. This appendix is intended as a review of matrix algebra, rather than a detailed treatment of it.

A.1 Terminology and Notation

A **matrix** is a rectangular array of numbers. A **vector** is a matrix with only one row (a **row vector** or only one column (a **column vector**).

The expression, "the (i,j) element of a matrix," will mean its element in row i, column j.

Please note the following conventions:

- Capital letters, e.g. A and X, will be used to denote matrices and vectors.
- Lower-case letters with subscripts, e.g. $a_{2,15}$ and x_8, will be used to denote their elements.
- Capital letters with subscripts, e.g. A_{13}, will be used to denote submatrices and subvectors.

If A is a **square** matrix, i.e. one with equal numbers n of rows and columns, then its **diagonal** elements are a_{ii}, i = 1,...,n.

The **norm** (or **length**) of an n-element vector **X** is

$$\| X = \sqrt{\sum_{i=1}^{n} x_i^2} \tag{A.1}$$

A.1.1 Matrix Addition and Multiplication

- For two matrices have the same numbers of rows and same numbers of columns, addition is defined elementwise, e.g.

$$\begin{pmatrix} 1 & 5 \\ 0 & 3 \\ 4 & 8 \end{pmatrix} + \begin{pmatrix} 6 & 2 \\ 0 & 1 \\ 4 & 0 \end{pmatrix} = \begin{pmatrix} 7 & 7 \\ 0 & 4 \\ 8 & 8 \end{pmatrix} \tag{A.2}$$

- Multiplication of a matrix by a **scalar**, i.e. a number, is also defined elementwise, e.g.

$$0.4 \begin{pmatrix} 7 & 7 \\ 0 & 4 \\ 8 & 8 \end{pmatrix} = \begin{pmatrix} 2.8 & 2.8 \\ 0 & 1.6 \\ 3.2 & 3.2 \end{pmatrix} \tag{A.3}$$

- The **inner product** or **dot product** of equal-length vectors X and Y is defined to be

$$\sum_{k=1}^{n} x_k y_k \tag{A.4}$$

- The product of matrices A and B is defined if the number of rows of B equals the number of columns of A (A and B are said to be **conformable**). In that case, the (i,j) element of the product C is defined to be

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \tag{A.5}$$

  For instance,

$$\begin{pmatrix} 7 & 6 \\ 0 & 4 \\ 8 & 8 \end{pmatrix} \begin{pmatrix} 1 & 6 \\ 2 & 4 \end{pmatrix} = \begin{pmatrix} 19 & 66 \\ 8 & 16 \\ 24 & 80 \end{pmatrix} \tag{A.6}$$

  It is helpful to visualize c_{ij} as the inner product of row i of A and column j of B, e.g. as shown in bold face here:

$$\begin{pmatrix} \mathbf{7} & \mathbf{6} \\ 0 & 4 \\ 8 & 8 \end{pmatrix} \begin{pmatrix} \mathbf{1} & 6 \\ \mathbf{2} & 4 \end{pmatrix} = \begin{pmatrix} \mathbf{7} & 70 \\ 8 & 16 \\ 8 & 80 \end{pmatrix} \tag{A.7}$$

- Matrix multiplicatin is associative and distributive, but in general not commutative:

$$A(BC) = (AB)C \tag{A.8}$$

$$A(B + C) = AB + AC \tag{A.9}$$

$$AB \neq BA \tag{A.10}$$

A.2 Matrix Transpose

- The transpose of a matrix A, denoted A' or A^T, is obtained by exchanging the rows and columns of A, e.g.

$$\begin{pmatrix} 7 & 70 \\ 8 & 16 \\ 8 & 80 \end{pmatrix}' = \begin{pmatrix} 7 & 8 & 8 \\ 70 & 16 & 80 \end{pmatrix} \tag{A.11}$$

- If A + B is defined, then

$$(A + B)' = A' + B' \tag{A.12}$$

- If A and B are conformable, then

$$(AB)' = B'A' \tag{A.13}$$

A.3 Linear Independence

Equal-length vectors X_1,...,X_k are said to be **linearly independent** if it is impossible for

$$a_1X_1 + ... + a_kX_k = 0 \tag{A.14}$$

unless all the a_i are 0.

A.4 Determinants

Let A be an nxn matrix. The definition of the determinant of A, det(A), involves an abstract formula featuring permutations. It will be omitted here, in favor of the following computational method.

Let $A_{-(i,j)}$ denote the submatrix of A obtained by deleting its ith row and jth column. Then the determinant can be computed recursively across the kth row of A as

$$det(A) = \sum_{m=1}^{n} (-1)^{k+m} det(A_{-(k,m)}) \tag{A.15}$$

where

$$det \begin{pmatrix} s & t \\ u & v \end{pmatrix} = sv - tu \tag{A.16}$$

A.5 Matrix Inverse

- The **identity** matrix I of size n has 1s in all of its diagonal elements but 0s in all off-diagonal elements. It has the property that AI = A and IA = A whenever those products are defined.
- The A is a square matrix and AB = I, then B is said to be the **inverse** of A, denoted A^{-1}. Then BA = I will hold as well.
- A^{-1} exists if and only if its rows (or columns) are linearly independent.
- A^{-1} exists if and only if $det(A) \neq 0$.
- If A and B are square, conformable and invertible, then AB is also invertible, and

$$(AB)^{-1} = B^{-1}A^{-1} \tag{A.17}$$

A.6 Eigenvalues and Eigenvectors

Let A be a square matrix.[1]

[1]For nonsquare matrices, the discussion here would generalize to the topic of **singular value decomposition**.

- A scalar λ and a nonzero vector X that satisfy

$$AX = \lambda X \tag{A.18}$$

are called an **eigenvalue** and **eigenvector** of A, respectively.

- A matrix U is said to be **orthogonal** if its rows have norm 1 and are orthogonal to each other, i.e. their inner product is 0. U thus has the property that $UU' = I$ i.e. $U^{-1} = U$.

- If A is symmetric and real, then it is **diagonalizable**, i.e there exists an orthogonal matrix U such that

$$U'AU = D \tag{A.19}$$

for a diagonal matrix D. The elements of D are the eigenvalues of A, and the columns of U are the eigenvectors of A.