40 P

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Artificial Intelligence
Memo. No. 167

October 1968

Linear Separation and Learning

Marvin Minsky

and

Seymour Papert

This is a reprint of page proofs of Chapter 12 of Perceptrons, M. Minsky and S. Papert, MIT Press 1968, (we hope). It replaces A.I. Memo. No. 156 dated March 1968.

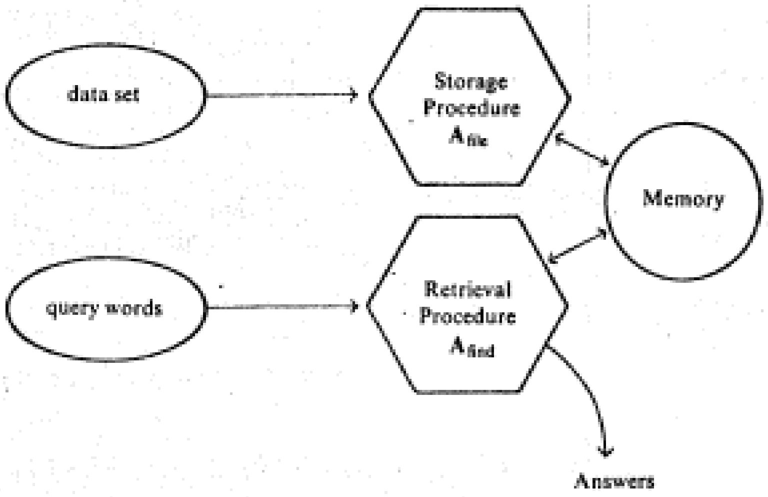
Linear Separation and Learning
12

12.0 Introduction

The perceptron and the convergence theorems of Chapter 11 are related to many other procedures that are studied in an extensive and disorderly literature under such titles as LEARNING MACHINES, MODELS OF LEARNING, INFORMATION RETRIEVAL, STATISTICAL DECISION THEORY, PATTERN RECOGNITION and many more. In this chapter we will study a few of these to indicate points of contact with the perceptron and to reveal deep differences. We can give neither a fully rigorous account nor a unifying theory of these topics: this would go as far beyond our knowledge as beyond the scope of this book. The chapter is written more in the spirit of inciting students to research than of offering solutions to problems.

12.1 Information Retrieval and Inductive Inference

The perceptron training procedures (Chapter 11) could be used to construct a device that operates within the following pattern of behavior:


data set
Storage Procedure A_{file}
Memory
query words
Retrieval Procedure A_{find}
Answers

During a "filing" phase, the machine is shown a "data set" of n-dimensional vectors—one can think of them as n-bit binary numbers or as points in n-space. Later, in a "finding" phase, the machine must be able to decide which of a variety of "query" vectors belong to the data set. To generalize this pattern we will

no dwg

188mh [illegible]

use the term A_{file} for an algorithm that examines elements of a data set to modify the information in a memory. A_{file} is designed to prepare the memory for use by another procedure, A_{find}, which will use the information in the memory to make decisions about query vectors.

This chapter will survey a variety of instances of this general scheme. We will begin by relating the PERCEPTRON procedure to the simplest such scheme: in the COMPLETE STORAGE procedure A_{file} merely copies the data vectors, as they come, into the memory. For each query vector, A_{find} searches exhaustively through memory to see if it is recorded there.

12.1.1 Comparing PERCEPTRON with COMPLETE STORAGE

Our purpose is to illustrate, in this simple case, some of the questions one might ask to compare retrieval schemes:

Is the procedure universal? The PERCEPTRON scheme works perfectly only under the restriction that the data set is linearly separable. COMPLETE STORAGE is universal: it works for any data set.

How much memory is required? COMPLETE STORAGE needs a memory large enough to hold the full data set. PERCEPTRON, when it is applicable, sometimes has a summarizing effect, in that the information capacity needed to store its coefficients $\{\alpha_i\}$ is substantially less than that needed to store the whole data set. We have seen (§10.2) that this is not generally true; the coefficients for ψ_{PARITY} may need much more storage than does the list of accepted vectors.

How quickly does A_{find} operate? The retrieval scheme—exhaustive search—specified for COMPLETE STORAGE is very slow (usually slower than PERCEPTRON's A_{find}, which must also retrieve all its coefficients from memory). On the other hand, very similar procedures could be much faster. For example, if A_{file} did not just store the data set in its order of entry, but *sorted* the memory into numerical order, then A_{find} could use a binary search, reducing the query-answer time to

$$\log_2(|\text{data set}|)$$

[illegible]

memory references. We shall study (in §12.6) $\mathbf{A}_{\text{file}}$ algorithms that sacrifice memory size to obtain dramatic further increases in speed (by the so-called *hash-coding* technique).

Can the procedure operate with some degree (perhaps measured probabilistically) of success even when $\mathbf{A}_{\text{file}}$ *has seen only a subset of the data set—call it a "data sample"?* PERCEPTRON might, but the COMPLETE STORAGE algorithm, as described, cannot make a reasonable guess when presented with a query vector not in the data sample. This deficiency suggests an important modification of the complete storage procedure: let $\mathbf{A}_{\text{find}}$, instead of merely checking whether the query vector is in the data sample, find that member of the data sample *closest* to it. This would lead, on an *a priori* assumption about the "continuity" of the data set, to a degree of generalization as good as the perceptron's. Unfortunately the speed-up procedures such as hash-coding cease to be available and we conjecture (in a sense to be made more precise in §12.7.6) that the loss is irremediable.

Other considerations we shall mention concern the operation of $\mathbf{A}_{\text{file}}$. We note that the PERCEPTRON and the COMPLETE STORAGE procedures share the following features:

They act incrementally, that is, change the stored memory slightly as a function of the currently presented member of the data set.

They operate in "real time" without using large quantities of auxiliary scratch-pad memory.

They can accept the data set in any order and are tolerant of repetitions that cause only delay but do not change the final state.

On the other hand they differ in at least one very fundamental way:

The perceptron's $\mathbf{A}_{\text{file}}$ is a "search procedure" based on feedback from its own results. The complete storage file algorithm is passive. The advantage for the perceptron is that under some conditions it finds an economical summarizing representation. The cost is that it may need to see each data point many times.

12.1.2 Multiple Classification Procedures

It is a slight generalization of these ideas to suppose the data set divided into a number of classes $\mathbf{F}_1, \ldots, \mathbf{F}_k$. The algorithm

[illegible]

$\mathbf{A}_{\text{file}}$ is presented as before with members of the data set but also with indications of the corresponding class. It constructs a body of stored information which is handed over to $\mathbf{A}_{\text{find}}$ whose task is to assign query points to their classes using this information.

Example: We have seen (§11.3.1) how to extend the concept of the perceptron to a multiple classification. The training algorithm, $\mathbf{A}_{\text{file}}$, finds k vectors $\mathbf{A}_1, \ldots, \mathbf{A}_k$, and $\mathbf{A}_{\text{find}}$ assigns the vector $\mathbf{\Phi}$ to $\mathbf{F}_j$ if

$\mathbf{\Phi} \cdot \mathbf{A}_j > \mathbf{\Phi} \cdot \mathbf{A}_i \quad (\text{all } i \neq j).$ INNER PRODUCT

Example: The following situation will seem much more familiar to many readers. If we think of each class $\mathbf{F}_j$ as a "clump" or "cloud" or "cluster" of points in $\mathbf{\Phi}$-space, then we can imagine that with each $\mathbf{F}_j$ is associated a special point $\mathbf{B}_j$ that is, somehow, a "typical" or "average" point. For example, $\mathbf{B}_j$ could be the *center of gravity*, that is, the *mean* of all the vectors in $\mathbf{F}_j$ (or, say, of just those vectors that so far have been observed to be in $\mathbf{F}_j$). Then a familiar procedure is: $\mathbf{\Phi}$ is judged to be in that $\mathbf{F}_j$ for which the Euclidean distance

$|\mathbf{\Phi} - \mathbf{B}_j|.$

is the *smallest*. That is, each $\mathbf{\Phi}$ is identified with the nearest $\mathbf{B}$-point.

Now this nearness scheme and the inner-product scheme might look quite different, but they are essentially the same! For we have only to observe that the set of points closer to one given point $\mathbf{B}_1$ than to another $\mathbf{B}_2$ is bounded by a hyperplane (Figure 12.1), and hence can be defined by a linear inequal-

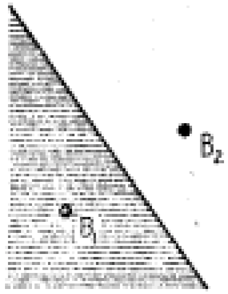


Figure 12.1

[illegible]

ity. Similarly, the points closest to one of a number of $\mathbf{B}_j$'s form a (convex) polygon (Figure 12.2) and this is true in higher dimensions, also.

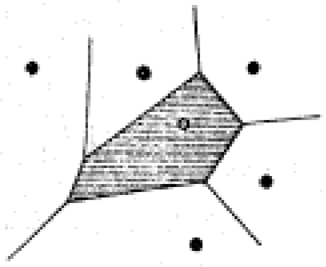


Figure 12.2

Formally, we see this by observing that

$$|\mathbf{\Phi} - \mathbf{B}_j|^2 = |\mathbf{\Phi}|^2 - 2\mathbf{\Phi}\cdot\mathbf{B}_j + |\mathbf{B}_j|^2.$$

Now, if we can assume that all the Φ's have the same length L then the Euclidean distance (B) will be *smallest* when

$$\mathbf{\Phi}\cdot\mathbf{B}_j - \tfrac{1}{2}|\mathbf{B}_j|^2 = \mathbf{\Phi}\cdot\mathbf{B}_j - \theta_j$$

is largest. But this has exactly the inner-product, if the "threshold" is removed by §1.2.1 (1). To see that the inner-product concept loses nothing by requiring the Φ's to have the same length, we add an extra dimension and replace each $\mathbf{\Phi} = [\varphi_1, \ldots, \varphi_n]$ by

$$\hat{\mathbf{\Phi}} = \left[\varphi_1, \ldots, \varphi_n, \left(n^2 - \sum_1^n \varphi_i\right)^{1/2}\right]$$

so that all $\hat{\Phi}$'s have length $L^2 = n$. We have to add one dimension to the $\mathbf{B}$'s, too, but can always set its coefficient to zero.

12.2 A Variety of Classification Algorithms

We select, from the infinite variety of schemes that one might use to divide a space into different classes, a few schemes that illustrate aspects of our main subject: computation and linear separation. We will summarize each very briefly here; the remainder of the chapter compares and contrasts some aspects of their algorithmic structures, memory requirements, and commitments they make about the nature of the classes.

[illegible]

Each of our models uses the same basic form of decision algorithm for $\mathbf{A}_{\text{find}}$. In each case there is assigned to each class $\mathbf{F}_j$ one or more vectors $\mathbf{A}_i$; we will represent this assignment by saying that $\mathbf{A}_i$ is associated with $\mathbf{F}_{j(i)}$. Given a vector $\mathbf{\Phi}$, the decision rule is always to choose that $\mathbf{F}_{j(i)}$ for which $\mathbf{A}_i \cdot \mathbf{\Phi}$ is largest. As noted in §12.1.2 this is mathematically equivalent to a rule that minimizes $|\mathbf{\Phi} - \mathbf{A}_i|$ or some very similar formula.

For each model we must also describe the algorithm $\mathbf{A}_{\text{file}}$ that constructs the $\mathbf{A}_i$'s, on the basis of prior experience, or *a priori* information about the classes. In the brief vignettes below, the fine details of the $\mathbf{A}_{\text{file}}$ procedures are deferred to other sections.

12.2.1 The PERCEPTRON Procedure
There is one vector $\mathbf{A}_j$ for each class $\mathbf{F}_j$. $\mathbf{A}_{\text{file}}$ can be the procedures described in §11.1 for the 2-class case and in §11.4.1 for the multiclass case.

12.2.2 The BAYES Linear Statistical Procedure
Again we have one $\mathbf{A}_j$ for each $\mathbf{F}_j$. $\mathbf{A}_{\text{file}}$ is quite different, however. For each class $\mathbf{F}_j$ and each partial predicate φ_i, define

$$w_{ij} = \log\left(\frac{p_{ij}}{1 - p_{ij}}\right),$$

where p_{ij} is *the probability that* $\varphi_i = 1$, *given that* $\mathbf{\Phi}$ *is in* $\mathbf{F}_j$. Then define

$$\mathbf{A}_j = (\theta_j, w_{1j}, w_{2j}, \ldots).$$

We will explain in §12.4.3 the conditions under which this use of "probability" makes sense, and describe some "learning" algorithms that could be used to estimate or approximate the w_{ij}'s.

The BAYES procedure has the advantage that, provided certain statistical conditions are satisfied, it gives good results for classes that are *not* linearly separable. In fact it gives the lowest possible error rate for procedures in which $\mathbf{A}_{\text{file}}$ depends only on conditional probabilities, given that the φ's are statistically independent in the sense explained in §12.4.2. It is astounding that this is achieved by a linear formula.

[illegible]

12.2.3 The BEST PLANES Procedure

In different situations either PERCEPTRON or BAYES may be superior. But often, when the $\mathbf{F}_j$'s are not linearly separable, there will exist a set of $\mathbf{A}_j$ vectors which will give fewer errors than either of these schemes. So define the BEST PLANES procedure to use that set of $\mathbf{A}_j$'s for which choice of the largest $\mathbf{A}_j \cdot \Phi$ gives the fewest errors.

By definition, BEST PLANES is always at least as good as BAYES or PERCEPTRON. This does not contradict the optimality of BAYES since the search for the best plane uses information other than the conditional probabilities. Unfortunately no practically efficient $\mathbf{A}_{\text{file}}$ is known for discovering its $\mathbf{A}_j$'s. As noted in §12.3, hill-climbing will apparently not work well because of the local peak problem.

12.2.4 The ISODATA Procedure

In the schemes described up to now, we assigned exactly one **A**-vector to each **F**-class. If we shift toward the minimum-distance viewpoint, this suggests that such procedures will work satisfactorily only when the **F**-classes are "localized" into relatively isolated, single regions—one might think of clumps, clusters, or clouds. Given this intuitive picture, one naturally asks what to do if an **F**-class, while not a neat spherical cluster, is nevertheless semilocalized as a small number of clusters or, perhaps, a snake-like structure. We could still handle such situations, using the least-distance $\mathbf{A}_{\text{find}}$, by assigning an **A**-vector to each subcluster of each **F**, and using several **A**'s to outline the spine of the snake. To realize this concept, we need an $\mathbf{A}_{\text{file}}$ scheme that has some ability to analyze distributions into clusters. We will describe one such scheme, called ISODATA, in §12.5.

12.2.5 The NEAREST NEIGHBOR Procedure

Our simplest and most radical scheme assumes no limit on the number of **A**-vectors. $\mathbf{A}_{\text{file}}$ stores in memory every Φ that has ever been encountered, together with the name of its associated **F**-class. Given a query vector Φ_0, we find which Φ in the memory is closest to Φ_0, and choose the **F**-class associated with that Φ.

This is a very generally powerful method: it is very efficient on many sorts of cluster configurations; it never makes a mistake on an already seen point; in the limit it approaches zero error

[illegible]

except under rather peculiar circumstances (one of which is discussed in the following section).

NEAREST NEIGHBOR has an obvious disadvantage—the very large memory required—and a subtle disadvantage: there is reason to suspect that it entails large, and fundamentally unavoidable, computation costs (discussed in §12.6).

12.3 Heuristic Geometry of Linear Separation Methods

The diagrams of this section attempt to suggest some of the behavioral aspects of the methods described in §12.4. To compensate for our inability to depict multidimensional configurations, we use two-dimensional multivalued coordinates. The diagrams may appear plausible, but they are really defective images that do not hint at the horrible things that can happen in a space of many dimensions.

Using this metaphorical kind of picture, we can suggest two kinds of situations which tend to favor one or the other of BAYES or PERCEPTRON (see Figure 12.3).

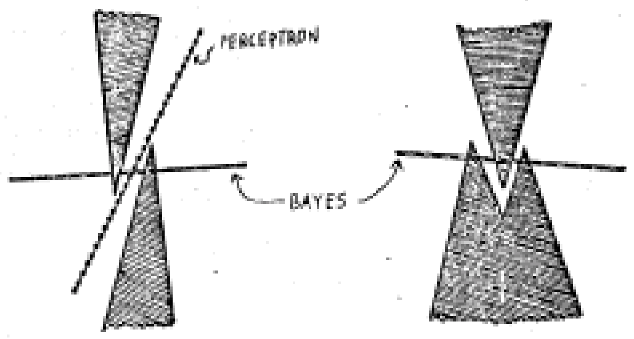


Figure 12.3

The BAYES line in Figure 12.3 tends to lie perpendicular to the line between the "mean" points of $\mathbf{F}_-$ and $\mathbf{F}_+$. Hence in Figure 12.3(a), we find that BAYES makes some errors. The sets are, in fact, linearly separable, hence PERCEPTRON, eventually, makes no errors at all. In Figure 12.3(b) we find that BAYES makes a few errors, just as in 12.3(a). We don't know much about PERCEPTRON in nonseparable situations; it is clear that in some situations it

[illegible]

will not do as well as BAYES. By definition BEST PLANE, of course, does at least as well as either BAYES or PERCEPTRON.

From the start, the very suggestion that any of these procedures will be any good at all amounts to an *a priori* proposal that the **F**-classes can be fitted into simple clouds of some kind, perhaps with a little overlapping, as in Figure 12.4. Such an assumption

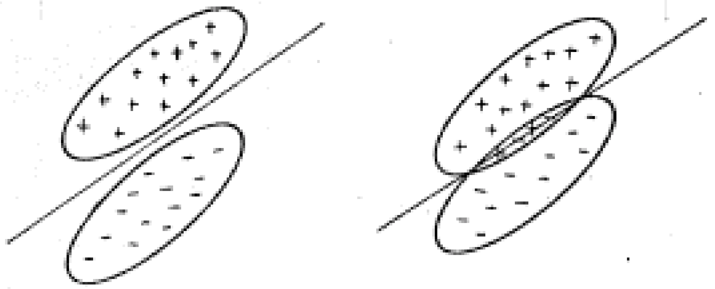


Figure 12.4

could be justified by some reason for believing that the difference between $\mathbf{F}_+$ and $\mathbf{F}_-$ are due to some one major influence plus a variety of smaller, secondary effects. In general PERCEPTRON tends to be sensitive to the outer boundaries of the clouds, and relatively insensitive to the density distributions inside, while BAYES weights all Φ's equally. In cases that do not satisfy either the single-cloud or the slight-overlap condition (Figure 12.5), we can expect BAYES to do badly, and presumably PERCEPTRON also. BEST PLANE can be substantially better because it is not subject to the bad influence of symmetry. But *finding* the BEST PLANE is

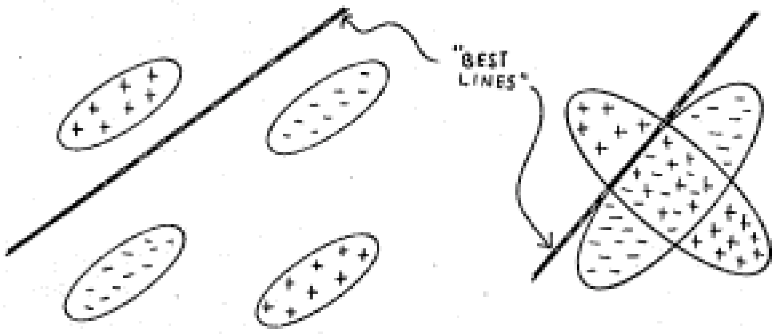


Figure 12.5

Ti-h

likely to involve bad computation problems because of multiple, locally optimal "hills." Figure 12.6 shows some of the local peaks for BEST PLANE in the case of a bad "paritylike" situation. Here, even ISODATA will do badly unless it is allowed to have one **A**-vector for nearly every clump. But in the case of a moderate number of clumps, with an $\mathbf{A}_i$ in each, ISODATA should do quite well. (See §12.5.) Generally, we would expect PERCEPTRON to be slightly better than BAYES because it exploits behavioral feedback, worse because of undue sensitivity to isolated errors.

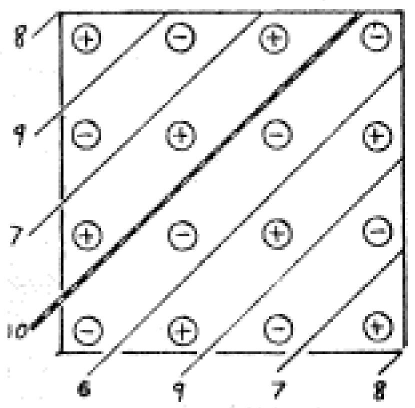
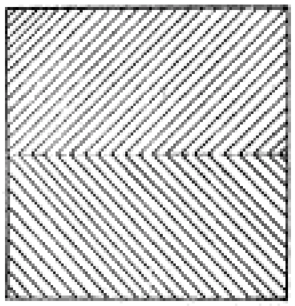


Figure 12.6

One would expect the NEAREST NEIGHBOR procedure to do well under a very wide range of conditions. Indeed, NEAREST NEIGHBOR in the limiting case of recording all Φ's with their class names, will do at least as well as any other procedure. There are conditions, though, in which NEAREST NEIGHBOR does not do so well until the sample size is nearly the whole space. Consider, for example, a space in which there are two regions:



$$P(\Phi \in F_+) = p$$

$$P(\Phi \in F_+) = 1 - p = q$$

In the upper region a fraction p of the points are in $\mathbf{F}_+$, and these are randomly distributed in space, and similarly for $\mathbf{F}_-$ in the lower region. Then if a small fraction of the points are already recorded, the probability that a randomly selected point has the same **F** as its nearest recorded neighbor is

$$p^2 + q^2 = 1 - 2pq,$$

while the probability of correct identification by BAYES or by BEST PLANE is simply p. Assuming that $p > \frac{1}{2}$ (if not, just exchange p and q) we see that

$$\text{Error}_{\text{BEST PLANE}} < \text{Error}_{\text{NEAREST NEIGHBOR}} < 2 \times \text{Error}_{\text{BEST PLANE}}$$

so that NEAREST NEIGHBOR is worse than BEST PLANE, but not arbitrarily worse. This effect will remain until so many points have been sampled that there is a substantial chance that the sampled point has been sampled before, that is, until a good fraction of the whole space is covered.

On the other side, to the extent that the "mixing" of $\mathbf{F}_+$ and $\mathbf{F}_-$ is less severe (see Figure 12.7), the NEAREST NEIGHBOR will converge to very good scores as soon as there is a substantial chance of finding *one* sampled point in most of the microclumps.

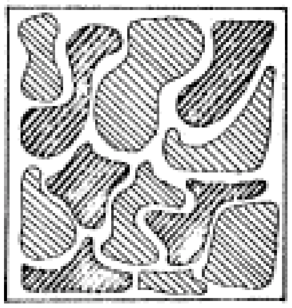


Figure 12.7

A very bad case is a paritylike structure in which NEAREST NEIGHBOR actually does worse than chance. Suppose that $\Phi \in \mathbf{F}_1$ if and only if $\varphi_i = 1$ for an even number of i's. Then, if there are n φ's, each Φ will have exactly n neighbors whose distance d

[illegible]

satisfies $0 < d \leq 1$. Suppose that all but a fraction q of all possible Φ's have already been seen. Then NEAREST NEIGHBOR will err on a given Φ if it has not been seen (probability $= q$) but one of its immediate neighbors has been seen (probability $= 1 - q^n$). So the probability of error is $\geq q(1 - q^n)$, which, for large n, can be quite near certainty.

This example is, of course, "pathological," as mathematicians like to say, and NEAREST NEIGHBOR is probably good in many real situations. Its performance depends, of course, on the precise "metric" used to compute distance, and much of classical statistical technique is concerned with optimizing coordinate axes and measurement scales for related applications.

Finally, we remark that because the memory and computation costs for this procedure are so high, it is subject to competition from more elaborate schemes outside the regime of linear separation—and hence outside the scope of this book.

12.4 Decisions Based on Probabilities of Predicate-Values

Some of the procedures discussed in previous sections might be called "statistical" in the weak sense that their success is not guaranteed except up to some probability. The procedures discussed in this section are statistical also in the firmer sense that they do not store members of the *data set* directly, but instead store statistical parameters, or measurements, about the data set. We shall analyze in detail a system that computes—or estimates—the conditional probabilities p_{ij} that, for each class $\mathbf{F}_j$ the predicate φ_i has the value 1. It stores these p_{ij}'s together with the absolute probabilities p_j of Φ being in each of the $\mathbf{F}_j$'s.

Given an observed Φ, the decision to choose an $\mathbf{F}_j$ is a typical statistical problem usually solved by a "maximum likelihood" or Bayes-rule method. It is interesting that procedures of this kind resemble very closely the perceptron separation methods. In fact, when we can assume that the conditional probabilities p_{ij} are suitably independent (§12.4.2) it turns out that the best procedure is the linear threshold decision we called BAYES in §12.2.2. We now show how this comes about.

12.4.1 Maximum Likelihood and Bayes Law

In Chapter 11 we assumed that each Φ is associated with a unique $\mathbf{F}_j$. We now consider the slightly more general case in which the

[illegible]

same Φ could be produced by events in several different **F**-classes. Then, given an observed Φ we cannot in general be sure which $\mathbf{F}_j$ is responsible, but we can at best know the associated probabilities.

Suppose that a particular Φ_0 has occurred and we want to know which **F** is most likely. Now if $\mathbf{F}_j$ is responsible, then the "joint event" $\mathbf{F}_j \wedge \Phi_0$ has occurred; this has (by definition) probability $\mathcal{P}(\mathbf{F}_j \wedge \Phi_0)$. Now (by definition of conditional probability) we can write

$$\mathcal{P}(\mathbf{F}_j \wedge \Phi_0) = \mathcal{P}(\mathbf{F}_j)\cdot\mathcal{P}(\Phi_0 \mid \mathbf{F}_j). \tag{1}$$

That is, the probability that both $\mathbf{F}_j$ and Φ_0 will happen together is equal to the probability that $\mathbf{F}_j$ will occur multiplied by the probability that *if* $\mathbf{F}_j$ *occurs so will* Φ_0.

We should choose that $\mathbf{F}_j$ which gives Formula 1 the largest value because that choice corresponds to the most likely of those events that could have occurred;

$$\mathbf{F}_1 \wedge \Phi_0 \qquad \mathbf{F}_2 \wedge \Phi_0 \qquad \cdots \qquad \mathbf{F}_k \wedge \Phi_0.$$

These are serious practical obstacles to the direct use of formula 1. If there are many different Φ's it becomes impractical to store all the decisions in memory, let alone to estimate them all on the basis of empirical observation. Nor has the system any ability to guess about Φ's it has not seen before. We can escape all these difficulties by making one critical assumption—in effect, assuming the situation closely fits a certain model—that the partial predicates of $\Phi = (\varphi_1, \ldots, \varphi_m)$ are suitably independent.

12.4.2 Independence

Up to now we have suppressed the X's of earlier chapters because we did not care where the values of the φ's came from. We bring them back for a moment so that we can give a natural context to the independence hypothesis.

We can evade the problems mentioned above if we can assume that the tests $\varphi_i(X)$ are statistically independent over each **F**-class. Precisely, this means that for any $\Phi(X) = (\varphi_1(X), \ldots, \varphi_m(X))$ we can assert that, for each j,

$$\mathcal{P}(\Phi \mid \mathbf{F}_j) = \mathcal{P}(\varphi_1 \mid \mathbf{F}_j) \times \cdots \times \mathcal{P}(\varphi_m \mid \mathbf{F}_j). \tag{2}$$

[illegible]

We emphasize that this is a most stringent condition. For example, it is equivalent to the assertion that:

Given that a Φ *is in a certain* **F**-*class, if one is told also the values of some of the* φ*'s, this gives absolutely no further information about the values of the remaining* φ*'s.*

Experimentally, one would expect to find independence when the variations in the values of φ's are due to "noise" or measurement uncertainties within the individual φ-mechanisms (Figure 12.8).

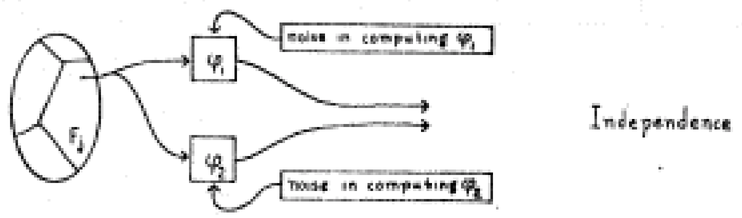

noise in computing φ_1
φ_1
F_j
φ_2
noise in computing φ_2
Independence

Figure 12.8

For, to the extent that these have separate causes, one would not expect the values of one to help predict the values of another. But if the variations in the φ's are *due to selection of different* X*'s from the same* **F**-*class*, one would *not* ordinarily assume independence, since the value of each φ tells us something about which X in **F** has occurred, and hence should help at least partly to predict the values of other φ's (see Figure 12.9).

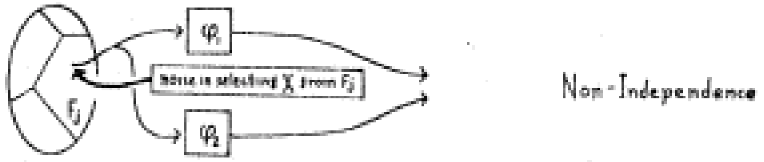


Figure 12.9

An extreme example of nonindependence is the following: there are two classes, $\mathbf{F}_1$ and $\mathbf{F}_2$, and two φ's, defined by

$$\begin{cases} \varphi_1(X) = \text{a pure random variable with } \mathcal{P}(\varphi_1(X) = 1) = \frac{1}{2}. \\ \qquad\qquad \text{Its value is determined by tossing a coin, not by } X. \\ \varphi_2(X) = \begin{cases} \varphi_1(X) \text{ if } X \in \mathbf{F}_1, \\ 1 - \varphi_1(X) \text{ if } X \in \mathbf{F}_2. \end{cases} \end{cases}$$

Then $\mathcal{P}(\varphi_1 \wedge \varphi_2 | F_1) = \frac{1}{2}$.
But $\mathcal{P}(\varphi_1 | F_1) \cdot \mathcal{P}(\varphi_2 | F_1) = \frac{1}{2} \cdot \frac{1}{2}$.

Notice that *neither φ_1 nor φ_2 taken alone gives any information whatever about* **F**! Each appears to be a random coin toss. But from both one can determine perfectly which **F** has occured,

for $\varphi_1 = \varphi_2$ implies $\mathbf{F}_1$,
while $\varphi_1 \neq \varphi_2$ implies $\mathbf{F}_2$

with absolute certainty.

REMARK: We assume only independence *within* each class $\mathbf{F}_j$. So if X is not given, then learning one φ-value *can* help predict another. For example, suppose that

$\varphi_1 = \varphi_2 = 0$ if $X \in \mathbf{F}_1$,
$\varphi_1 = \varphi_2 = 1$ if $X \in \mathbf{F}_2$.

These two φ's are in fact independent on each **F**. But if we *did not know in advance that $X \in$* $\mathbf{F}_1$ but were told that $\varphi_1 = 0$, we could indeed then predict that $\varphi_2 = 0$ also, without this violating our independence assumption. (If we had previously been told that $X \in \mathbf{F}_1$, then we could *already* predict the value of φ_2; in that case learning the value of φ_1 would have no effect on our prediction of φ_2.)

12.4.3 The maximum likelihood decision, for independent φ's, is a linear threshold predicate!

Assume that the φ_i's are statistically independent for each $\mathbf{F}_j$. Define

$p_j = \mathcal{P}(\mathbf{F}_j)$,
$p_{ij} = \mathcal{P}(\varphi_i = 1 | \mathbf{F}_j)$,
$q_{ij} = 1 - p_{ij} = \mathcal{P}(\varphi_i = 0 | \mathbf{F}_j)$.

Suppose that we have just observed a $\Phi = (\varphi_1, \ldots, \varphi_m)$, and we want to know which $\mathbf{F}_j$ was most probably responsible for this. Then, according to Formulas 1 and 2, we will choose that j which maximizes

$$p_j \cdot \prod_{\varphi_i = 1} p_{ij} \cdot \prod_{\varphi_i = 0} q_{ij}$$

[illegible]

$$= p_j \cdot \prod_i p_{ij}^{\varphi_i} \cdot q_{ij}^{(1-\varphi_i)}$$

$$= p_j \cdot \prod_i \left(\frac{p_{ij}}{q_{ij}}\right)^{\varphi_i} \cdot \prod_i q_{ij}.$$

Because sums are more familiar to us than products, we will replace these by their logarithms. Since log x increases when x does, we still will select the largest of

$$\sum_i \varphi_i \cdot \log \frac{p_{ij}}{q_{ij}} + \left(\log p_j + \sum_i \log q_{ij}\right). \tag{3}$$

Because the right-hand expression is a constant that depends only upon j, and not upon the experimental Φ, all of Formula 3 can be written simply as

$$\Sigma\, w_{ij}\varphi_i + \theta_j. \tag{3'}$$

Example 1: In the case that there are just two classes, $\mathbf{F}_1$ and $\mathbf{F}_2$, we can decide that $X \in \mathbf{F}$ whenever

$$\Sigma\, w_{i1}\varphi_i + \theta_1 > \Sigma\, w_{i2}\varphi_i + \theta_2,$$

that is, when

$$\Sigma (w_{i1} - w_{i2})\varphi_i > (\theta_2 - \theta_1), \tag{4}$$

which has the form of a linear threshold predicate

$$\psi = [\Sigma\, \alpha_i \varphi_i > \theta].$$

Thus we have the remarkable result that the hypothesis of independence among the φ's leads directly to the familiar linear decision policy.

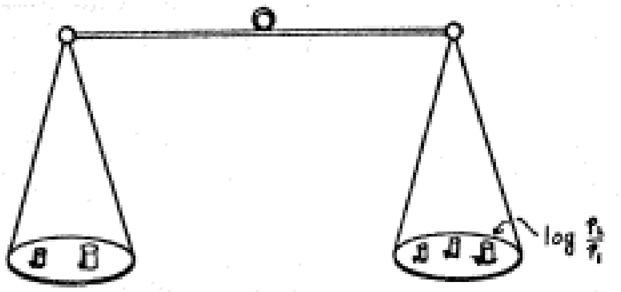
Example 2 (probabilities of error): Suppose that for all i, $p_{i1} = q_{i2}$. Then p_{i1} is the probability that $\varphi_i(X) = \psi(X)$ and q_{i1} is the probability that $\varphi_i(X) \neq \psi(X)$, that is, that φ_i makes an error in (individually) predicting the value of $\psi = [X \in \mathbf{F}_1]$.

[illegible]

Then inequality 4 becomes

$$\sum_i w_{i1}(2\varphi_i - 1) > \log \frac{p_2}{p_1}. \tag{4'}$$

Now observe that the $(2\varphi_i - 1)$ term has the effect of *adding* or *subtracting* w_{i1} according to whether $\varphi_i = 1$ or 0. Thus, we can think of the w's as weights to be added (according to the φ's) to one side or the other of a balance:



The $\log(p_2/p_1)$ is the "*a priori* weight" in favor of $\mathbf{F}_2$ at the start, and each $w_{i1} = \log(p_{i1}/q_{i1})$ is the "weight of the evidence" that $\varphi_i = 1$ gives in favor of $\mathbf{F}_1$.

It is quite remarkable that the optimal separation algorithm—given that the φ-probabilities are independent—has the form (inequality 4) of a linear threshold predicate. But one must be sure to understand that if $[\Sigma\, \alpha_\varphi \varphi > \theta]$ is the "optimal" predicate obtained by the independent-probability method, yet does *not* perfectly realize a predicate ψ, this does not preclude the existence of a precise separation $[\Sigma\, \alpha'_\varphi \varphi > \theta']$ which always agrees with ψ. [This is the situation suggested by Figure 12.3(a).] For inequality 4 is "optimal" only in relation to all $\mathbf{A}_{\text{Bk}}$ procedures *that use no information other than the conditional probabilities* $\{p_j\}$ and $\{p_{ij}\}$, while a perceptron computes coefficients by a nonstatistical search procedure that is sensitive to individual events.

Thus, if ψ is in fact in $L(\Phi)$ the perceptron will eventually perform at least as well as any linear-statistical machine. The latter family can have the advantage in some cases:

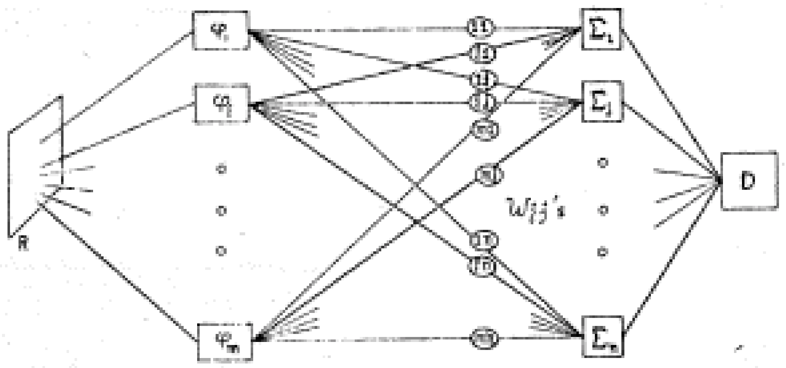
1. If $\psi \notin L(\Phi)$ the statistical scheme may produce a good approximate separation while the perceptron might fluctuate wildly.

2. The time to achieve a useful level may be long for the perceptron file algorithm which is basically a serial search procedure. The linear-statis-

tical machine is basically more parallel, because it finds each coefficient independently of the others, and needs only a fair sample of the **F**'s. (While superficially perceptron coefficients appear to be changed individually, each decision about a change depends upon a test involving all the coefficients.)

12.4.4 Layer-Machines

Formula 3′ suggests the design of a machine for making our decision:

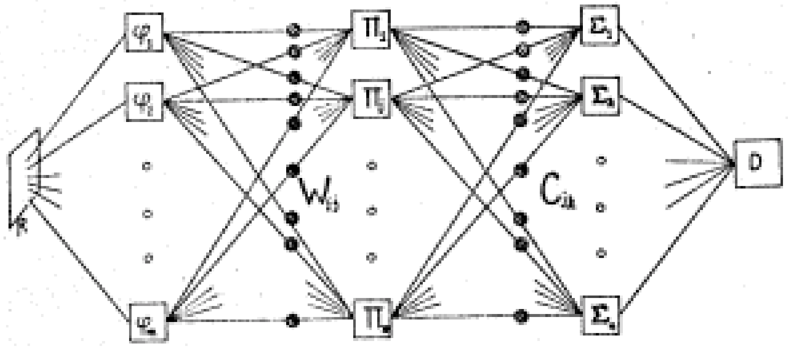


D is a device that simply decides which of its inputs is the largest. Each φ-device emits a standard-sized pulse [if $\varphi(X) = 1$] when X is presented. The pulses are multiplied by the w_{ij} quantities as indicated, and summed at the Σ-boxes. The θ_j terms may be regarded as corrections for the extent to which the p_{ij}'s deviate from a central value of $\frac{1}{2}$, combined with the *a priori* bias concerning $\mathbf{F}_j$ itself.

It is often desirable to minimize the *costs* of errors, rather than simply the chance of error. If we define C_{jk} to be the cost of guessing $\mathbf{F}_k$ when it is really $\mathbf{F}_j$ that has occurred, then it is easy to show that Formulas 1 and 2 now lead to finding the k that minimizes

$$\sum_j C_{jk} \cdot B_j \cdot \prod_i \left(\frac{p_{ij}}{q_{ij}}\right)^{\varphi_i},$$

where $B_j = \Pi\, q_{ij}$. It is interesting that this more complicated procedure also lends itself to the multilayer structure:



It ought to be possible to devise a training algorithm to optimize the weights in this using, say, the magnitude of a reinforcement signal to communicate to the net the cost of an error. We have not investigated this.

12.4.5 Probability-estimation procedures

The $\mathbf{A}_{\text{fik}}$ algorithm for the BAYES linear-statistical procedure has to compute, or estimate, either the probabilities p_{ij} and p_j of formula 3 or other statistical quantities such as "weight of evidence" ratios $p/(1-p)$. Normally these cannot be calculated directly (because they are, by definition, limits) so one must find *estimators*. The simplest way to estimate a probability is to find the ratio H/N of the number H of "favorable" events to the number N of all events in a sequence. If $\varphi^{[t]}$ is the value of φ on the tth trial, then an estimate of $\mathcal{P}(\varphi = 1)$ after n trials can be found by the procedure:

```
START:     Set $\alpha$ to 0.
           Set $n$ to 1.
REPEAT:    Set $\alpha$ to $\frac{(n-1)\alpha + \varphi^{[n]}}{n}$.
           Set $n$ to $n + 1$.
           Go to REPEAT.
```

which can easily be seen to recompute the "score" H/N after each event.

This procedure has the disadvantage that it has to keep a record of n, the number of trials. Since n increases beyond bound, this would require unlimited memory. To avoid this, we rewrite the

[illegible]

above program's computation in the form

$$\alpha^{[n]} = \left(1 - \frac{1}{n}\right)\alpha^{[n-1]} + \frac{1}{n}\varphi^{[n]}.$$

This suggests a simpler heuristic substitute: define

$$\begin{cases} \alpha^{[0]} = 0, \\ \alpha^{[n]} = (1-\epsilon)\alpha^{[n-1]} + \epsilon\cdot\varphi^{[n]}, \end{cases} \tag{5}$$

where ϵ is a "small" number between 0 and 1. It is easy to show that as n increases the *expected* or *mean* value of $\alpha^{[n]}$, which we will write as $\langle\alpha^{[n]}\rangle$, approaches p (that is, $\langle\varphi\rangle$) as a limit. For

$$\begin{aligned}\langle\alpha^{[1]}\rangle &= (1-\epsilon)\langle\alpha^{[0]}\rangle + \epsilon\langle\varphi^{[1]}\rangle = \epsilon p \\ &= [1-(1-\epsilon)]p,\end{aligned}$$

and

$$\begin{aligned}\langle\alpha^{[2]}\rangle &= (1-\epsilon)(1-(1-\epsilon))p + \epsilon p \\ &= (1-(1-\epsilon)^2)p,\end{aligned}$$

and one can verify that, for all n,

$$\begin{aligned}\langle\alpha^{[n]}\rangle &= (1-(1-\epsilon)^n)p \\ &\to p. \qquad (\text{as } n \to \infty)\end{aligned}$$

Thus, process 5 gives an estimation of the probability that $\varphi = 1$. A more detailed analysis would show how the estimate depends on the events of the recent past, with the effect of ancient events decaying exponentially—with coefficients like $(1-\epsilon)^{(n-t)}$.

Because process 5 "forgets," it certainly does not make "optimal" use of its past experience; but under some circumstances it will be able to "adapt" to changing environmental statistics, which could be a good thing. As a direct consequence of the decay, our estimator has a peculiar property: its variance "σ^2" does not approach zero. In fact, one can show that, for process 5,

$$\sigma^2 \to p(1-p)\frac{\epsilon}{2-\epsilon}$$

[illegible]

and this, while not zero, will be very small whenever ϵ is. The situation is thus quite different from the H/N estimate—whose variance is $p(1 - p)/n$ and approaches zero as n grows.

In fact, we can use the variance to compare the two procedures: If we "equate" the variances

$$p(1 - p) \cdot \frac{\epsilon}{2 - \epsilon} \cong p(1 - p) \cdot \frac{1}{n},$$

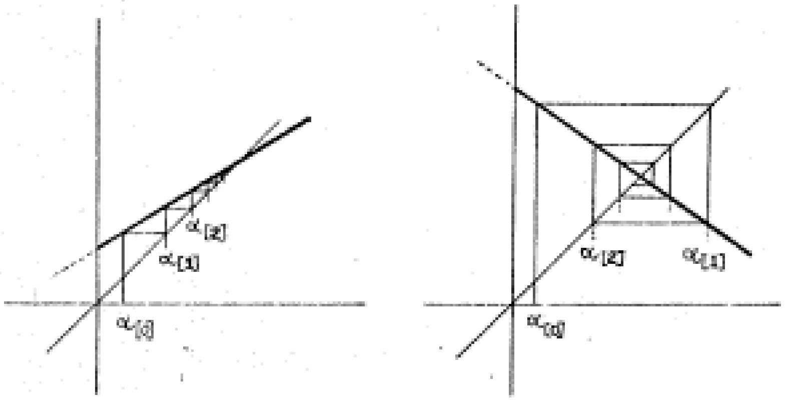
we obtain

$$n \sim \frac{2}{\epsilon},$$

suggesting that the reliability of the estimate of p given by process 5 is about the same as we would get by simple averaging of the last $2/\epsilon$ samples; thus one can think of the number $1/\epsilon$ as corresponding to a "time-constant" for forgetting.

Another estimation procedure one might consider is:

```
INITIALIZE:  Set $\alpha$ to anything.
REPEAT:      If $\varphi = 1$, set $\alpha$ to $\alpha + 1$.
             If $\varphi = 0$, set $\alpha$ to $(1 - \epsilon)\alpha$.
             Go to REPEAT.
```



Convergence to the Fixed-Point

[illegible]

or, equivalently, one could write

$$\alpha^{[n]} = (1 - \epsilon)\alpha^{[n-1]} + (1 + \epsilon\alpha^{[n-1]})\varphi^{[n]}.$$

It can be shown that this has an expected value, in the limit, of

$$\langle\alpha^{[n]}\rangle \to \frac{1}{\epsilon} \cdot \left(\frac{p}{1 - p}\right).$$

It is interesting that a direct estimate of the likelihood ratio is obtained by such a simple process as *if* $\varphi = 1$ *add* 1, *otherwise multiply by* $(1 - \epsilon)$. The variance, in case anyone cares, is

$$\sigma^2 = \frac{p}{(1 - p)^2} \cdot \frac{1}{1 - (1 - \epsilon)^2}.$$

12.4.6 The Samuel compromise

In his classical paper about "Some Studies in Machine Learning using the Game of Checkers," Arthur L. Samuel uses an ingenious combination of probability estimation methods. In his application it occasionally happens that *a new evidence term* φ_i *is introduced* (and an old one is abandoned because it has not been of much value in the decision process). When this happens there is a problem of preventing violent fluctuations, because after one or a few trials the term's probability estimate will have a large variance as compared with older terms that have better statistical records. Samuel uses the following algorithm to "stabilize" his system: he sets $\alpha^{[0]}$ to $\frac{1}{2}$ and uses

$$\alpha^{[n+1]} = \left(1 - \frac{1}{N}\right)\alpha^{[n]} + \frac{1}{N}\varphi^{[n+1]},$$

where N is set according to the "schedule":

$$N = \begin{cases} 16 & \text{if } n < 32, \\ 2^m & \text{if } 2^m \leq n < 2^{m+1} \text{ and } 32 \leq n \leq 256, \\ 256 & \text{if } 256 \leq n. \end{cases}$$

Thus, in the beginning the estimate is made as though the probability had already been estimated to be $\frac{1}{2}$ on the basis of several,

[illegible]

that is, the order of 16, trials. Then in the "middle" period, the algorithm approximates the uniform weighting procedure. Finally (when $n \sim 256$) the procedure changes to the exponential decay mode, with fixed N, so that recent experience can outweigh earlier results. (The use of the powers of two represents a convenient computer-program technique for doing this.)

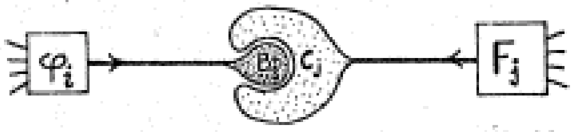
In Samuel's system, the terms actually used have the form we found in inequality 4' of §12.4.3

$$2\varphi^{[r]} - 1$$

so that the "estimator" ranges in the interval $-1 \leq \rho^{[r]} \leq +1$ and can be treated as a "correlation coefficient."

12.4.7 A simple "synaptic" reinforcer theory

Let us make a simple "neuronal model." The model is to estimate $p_{ij} = P(\varphi_i \mid F_j)$, using only information about occurrences of $[\varphi_i = 1]$ and of $[\Phi \in F_j]$. Our model will have the following "anatomy":



The bag B_i contains a very high and constant concentration of a substance E. When φ_i or F_j occur—or "fire"—the walls of the corresponding bags B_i and/or C_j become "permeable" to E for a moment. If φ_i alone occurs, nothing really changes, because B_i is surrounded by the impermeable C_j. If F_j alone occurs, C_j loses some E by diffusion to the outside: in fact, if α is the amount of E in C_j it may be assumed (by the usual laws of diffusion and concentration) to lose some fraction ϵ of α:

$$\alpha' = (1 - \epsilon)\alpha \quad \text{if} \begin{cases} F_j \text{ occurs and} \\ \varphi_i = 0. \end{cases}$$

If *both* φ_i and F_j occur then approximately the same loss will occur from C_j. Simultaneously, an essentially constant amount b will be "injected" by diffusion from B_i to C_j. So

$$\alpha' = (1 - \epsilon)\alpha + b \quad \text{if} \begin{cases} F_j \text{ occurs and} \\ \varphi_i = 1. \end{cases}$$

[illegible]

(We can assume that b is constant because the concentration of E is very high in B_i compared to that in C_j. One can invent any number of similar variations.) In either case we get

$$\alpha' = (1 - \epsilon)\alpha + \varphi b$$

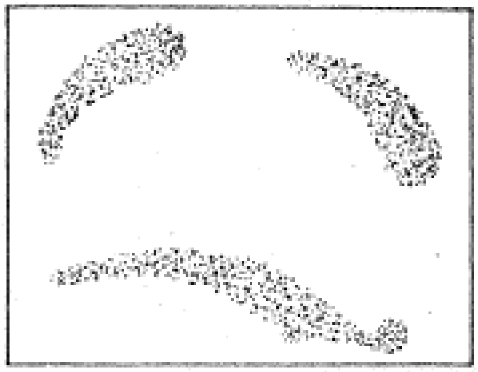
so that in the limit the mean of α approaches $b \cdot p$ (as can be seen from the analysis in §12.4.5). This is proportional to, and hence an estimator of $p_{ij} = P(\varphi_i \mid \mathbf{F}_j)$.

Thus the simple anatomy, combined with the membrane becoming permeable briefly following a nerve impulse, could give a quantity that is an estimator of the appropriate probability.

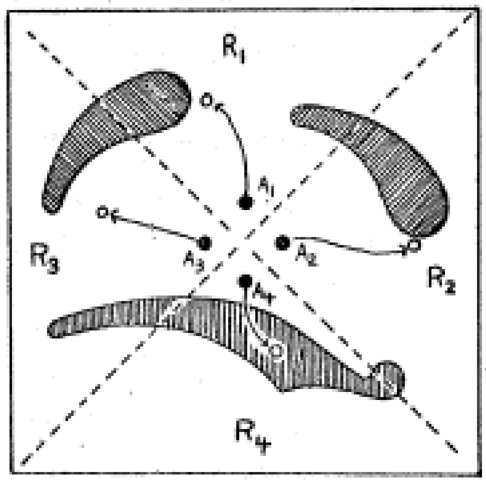
How could this representation of probability be translated into a useful neuronal mechanism? One could imagine all sorts of schemes: ionic concentrations--or rather, their logarithms!—could become membrane potentials, or conductivities, or even probabilities of occurrences of other chemical events. The "anatomy" and "physiology" of our model could easily be modified to obtain likelihood ratios. Indeed, it is so easy to imagine variants--the idea is so insensitive to details—that we don't propose it seriously, except as a family of simple yet intriguing models that a neural theorist should know about.

12.5 A_{file} algorithms for the ISODATA procedure

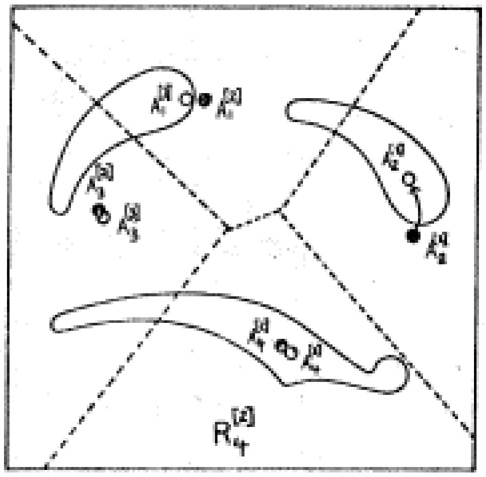
In this section we describe a procedure proposed by G. Ball and D. Hall to delineate "clusters" in an inhomogeneous distribution of vectors. The idea is best shown by a pictorial example: imagine a two-dimensional set of points $\{\Phi\}$ that fall into obvious clusters, like



Begin by placing a few "cluster-points" $\mathbf{A}_i^{[1]}$ into the space at some arbitrary locations, say, near the center. We then divide the set of Φ's into subsets $\mathbf{R}_i$, assigning each Φ to the *nearest* $\mathbf{A}_i^{[1]}$ point:

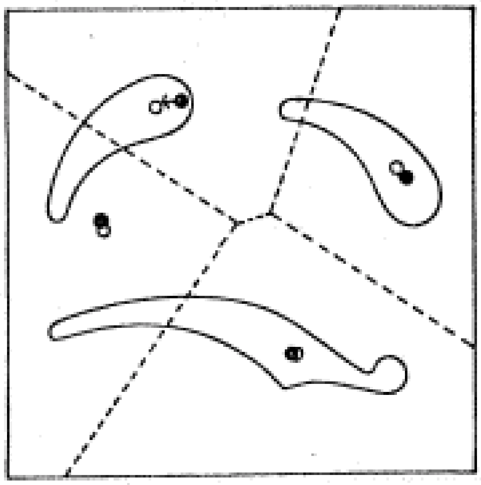


Next, we replace each $\mathbf{A}_i^{[1]}$ by a new cluster-point $\mathbf{A}_i^{[2]}$ which is the *mean* or center-of-gravity of the Φ's in $\mathbf{R}_i$, and then define $\mathbf{R}_i^{[2]}$ to be the set of Φ's nearest to $\mathbf{A}_i^2$:

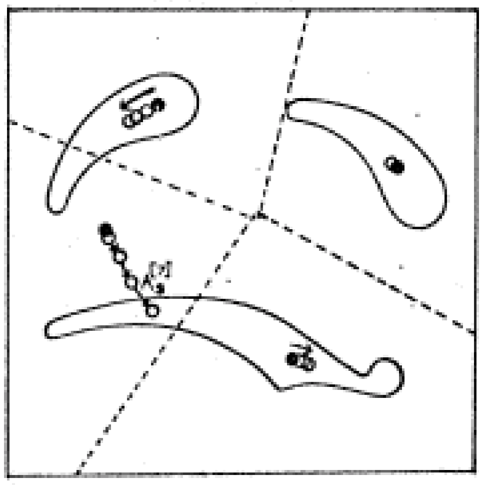


[illegible]

Repeating, we get a new set of $\mathbf{A}_i$'s and $\mathbf{R}_i$'s:



and next



From now on, there is little or no change; the cluster-points have "found the clusters."

Ball and Hall give a number of heuristic refinements for creating and destroying additional cluster points; for example, to add one if the variance of an **R**-set is "too large" and to remove one if two are "too close" together. Of course, one can usually "see" two-dimensional clusters by inspection, but ISODATA is alleged to give useful results in n-dimensional problems where "inspection" is out of the question.

To use this procedure, in our context, we need some way to combine its automatic classification ability with the information about the **F**-classes. An obvious first step would be to apply it separately to each **F**-class, and assign all the resulting **A**'s to that class. We do not know much about more sophisticated schemes that might lead to better results in the $\mathbf{A}_{\text{find}}$ stage.

12.5.1 An ISODATA convergence theorem

There is a theorem about ISODATA (told to us by T. Cover) that suggests that it leads to some sort of local minimum. Let us formalize the procedure by defining

$\mathbf{A}^{[n]}(\Phi)$ = the $\mathbf{A}_i^{[n]}$ that is nearest to Φ.

(If there are several equidistant nearest $\mathbf{A}_i$'s, choose the one with the smallest index.)

$\mathbf{R}_i^{[n]}$ = the set of Φ's for which $\mathbf{A}^{[n]}\Phi = \mathbf{A}_i^{[n]}$.

$\mathbf{A}_i^{[n+1]}$ = mean $\langle \mathbf{R}_i^{[n]} \rangle$.

Finally define a "score":

$$S^{[n]} = \sum_{\text{all } \Phi} |\Phi - \mathbf{A}^{[n]}(\Phi)|^2.$$

Theorem: $s^{[1]} > s^{[2]} > \ldots > s^{[n]} \ldots$

until there is no further change, that is, until $\mathbf{A}_i^{[n]} = \mathbf{A}_i^{[n+1]}$ for all i.

PROOF:

$$s^{[n]} = \sum_i \sum_{\mathbf{R}_i^{[n]}} |\Phi - \mathbf{A}_i^{[n]}|^2$$
$$> \sum_i \sum_{\mathbf{R}_i^{[n]}} |\Phi - \mathbf{A}_i^{[n+1]}|^2$$

because the mean ($\mathbf{A}_i^{[n+1]}$) of any set of vectors ($\mathbf{R}_i^{[n]}$) is just the point that minimizes the squared-distance sum to all the points (of $\mathbf{R}_i^{[n]}$). And this is, in turn,

$$\geq \sum_i \sum_{\mathbf{R}_i^{[n+1]}} |\Phi - \mathbf{A}_i^{[n+1]}|^2 = s^{[n+1]}$$

[illegible]

because each point will be transferred to an $\mathbf{R}_i^{[n+1]}$ for which the distance is minimal, that is, for all j,

$$|\Phi - \mathbf{A}_j^{[n+1]}| \geq |\Phi - \mathbf{A}_i^{[n]}|.$$

Corollary: *The sequence of decreasing positive numbers, $\{s^{[n]}\}$ approaches a limit. If there is only a finite set of Φ's, the $\mathbf{A}$'s must stop changing in a finite number of steps.*

For in the finite case there are only a finite number of partitions $\{\mathbf{R}_i\}$ possible.

12.5.2 Incremental methods

In analogy to the "reinforcement" methods in §12.4.5 we can approximate ISODATA by the following program:

```
START:     Choose a set of starting points $\mathbf{A}_i$.
REPEAT:    Choose a $\Phi$.
           Find $\mathbf{A}(\Phi)$: the $\mathbf{A}_i$ nearest to $\Phi$.
           Replace $\mathbf{A}(\Phi)$ by $(1 - \epsilon)\mathbf{A}(\Phi) + \epsilon \cdot \Phi$.
           Go to REPEAT.
```

It is clear that this program will lead to qualitatively the same sort of behavior; the A's will tend toward the *means* of their R-regions. But, just as in §12.4, the process will retain a permanent sampling-and-forgetting variance, with similar advantages and disadvantages. In fact, all the $\mathbf{A}_{\text{fix}}$ algorithms we have seen can be so approximated: there always seems to be a range from very local, incremental methods, to more accurate, somewhat less "adaptive" global schemes. We resume this discussion in §12.8.

12.6 Time vs. Memory for Exact Matching

Suppose that we are given a body of information—we will call it the *data set*—in the form of 2^a binary words each b digits in length (Figure 12.10); one can think of them as 2^a points chosen at random from a space of 2^b points. (Take a million $\cong 2^{20}$ words of length 100, for a practical example.) We will suppose that the data set is to be chosen at random from all possible sets so that one cannot expect to find much redundant structure within it. Then the ordered data set requires about $b \cdot 2^a$ bits of binary information for complete description. We won't, however, be in-

[illegible]

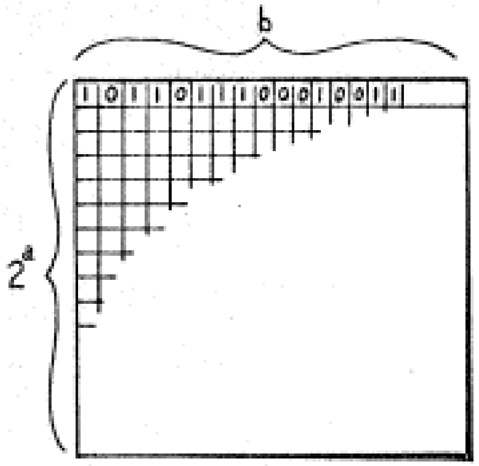


Figure 12.10

terested in the order of the words in the data set. This reduces the amount of information required to store the set to about $(b - a) \cdot 2^a$ bits.

We want a machine that, when given a random b-digit word w, will answer

QUESTION 1. Is w in the data set?*

and we want to formulate constraints upon how this machine works in such a way that we can separate computational aspects from memory aspects. The following scheme achieves this goal well enough to show, by examples, how little is known about the conjugacy between time and memory.

We will give our machine a memory of M separate bits—that is, one-digit binary words. We are required to compose—in advance, before we see the data set—two algorithms $\mathbf{A}_{\text{file}}$ and $\mathbf{A}_{\text{find}}$ that satisfy the following conditions:

1. $\mathbf{A}_{\text{file}}$ is given the data set. Using this as data, it fills the M bits of memory with information. Neither the data set nor $\mathbf{A}_{\text{file}}$ are used again, nor is $\mathbf{A}_{\text{find}}$ allowed to get any information about what $\mathbf{A}_{\text{file}}$ did, except by inspecting the contents of M.

* We will get to Question 2 in about fifteen minutes.

[illegible]

2. $\mathbf{A}_{\text{find}}$ is then given a random word, w, and asked to answer Question 1, using the information stored in the memory by $\mathbf{A}_{\text{file}}$. We are interested in how many bits $\mathbf{A}_{\text{find}}$ has to consult in the process.

3. The goal is to optimize the design of $\mathbf{A}_{\text{file}}$ and $\mathbf{A}_{\text{find}}$ to minimize the number of memory references in the question-answering computation, averaged over all possible words w.

12.6.1 Case 1: Enormous Memory
It is plausible that the larger be M, the smaller will be the average number of memory-references $\mathbf{A}_{\text{find}}$ must make. Suppose that

$$M \geq 2^b.$$

Let m_i be the ith bit in memory; then there is a bit m_w for each possible query word w, and we can define

$$\begin{cases} \mathbf{A}_{\text{file}}\text{: set } m_w \text{ to 1 if } w \text{ is in the data set} \\ \mathbf{A}_{\text{find}}\text{: } w \text{ is in the data set if } m_w = 1. \end{cases}$$

Thus, with a huge enough memory, only *one* reference is required to answer Question 1.

12.6.2 Case 2: Inadequate Memory
Suppose that

$$M < (b - a)2^a.$$

Here, the problem cannot be solved at all, since $\mathbf{A}_{\text{file}}$ cannot store enough information to describe the data set in sufficient detail.

12.6.3: Case 3: Binary Logarithmic Sort
Suppose that

$$M = b \cdot 2^a.$$

Now there is enough room to store the ordered data set. Define

$$\begin{cases} \mathbf{A}_{\text{file}}\text{: store the words of the data set in ascending numerical order.} \\ \mathbf{A}_{\text{find}}\text{: perform a binary search to see first which half of memory might contain } w\text{, then which quartile, etc.} \end{cases}$$

[illegible]

This will require at most $a = \log 2^a$ inspections of b-bit words, that is, $a \cdot b$ bit-inspections.

This is not an optimal search since, (1) one does not always need to inspect a *whole* word to decide which word to inspect next, and (2) it does not exploit the uniformity of distribution that the first a digits of the *ordered* data set will (on the average) show. Effect 1 reduces the required number from $a \cdot b$ to the order of $\frac{1}{2}a \cdot b$ and effect 2 reduces it from $a \cdot b$ to $a \cdot (b - a)$. We don't know exactly how these two effects combine.

12.6.4 Case 4: Exhaustive Search
Consider

$M = (b - a)2^a$.

This gives just about enough memory to represent the *unordered* data set. For example we could define

$\mathbf{A}_{\text{file}}$: First put the words of the data set in numerical order. Then compute their successive *differences*. These will require about $(b - a)$ bits each. Use a standard method of information theory, Huffman Coding (say), to represent this sequence; it will require about $(b - a)\,2^a$ bits.

But the only retrieval schemes we can think of are like

$\mathbf{A}_{\text{find}}$: Add up successive differences in memory until the sum equals or exceeds w. If equality occurs, then w is in the data set.

And this requires $\sim \frac{1}{2}(b - a)\,2^a$ memory references, on the average. It seems clear that, given this limit on memory, no $\mathbf{A}_{\text{file}}$ – $\mathbf{A}_{\text{find}}$ pair can do much better. That is, we suspect that

If no extra memory is available then, to answer Question 1, one must, on the average, search through half the memory.

One might go slightly further: even Huffman Coding needs some extra memory, and if there is none, $\mathbf{A}_{\text{file}}$ can only store an efficient "number" for the whole data set. Then the conjecture is that $\mathbf{A}_{\text{find}}$ must almost always look at almost all of memory.

[illegible]

12.6.5 Case 5: Hash Coding

Consider

$$M = b \cdot 2^a \cdot 2.$$

Here we have a case in which there is a substantial margin of extra memory—about twice what is necessary for the data set. The result here is really remarkable—one might even say counter-intuitive—because the mean number of references becomes *very* small. The procedure uses a concept well known to programmers who use it in "symbolic assembly programs" for symbol-table references, but does not seem to be widely known to other computer "specialists." It is called **hash-coding.**

There are many variants of this idea. We discuss a particular form adapted to a redundancy of two.

In the hash-coding procedure, $\mathbf{A}_{\text{file}}$ is equipped with a subprogram $R(w,j)$ that, given an integer j and a b-bit word w, produces an $(a + 1)$-bit word. The function $R(w,j)$ is "pseudorandom" in the sense that for each j, $R(w,j)$ maps the set of all 2^b input words with uniform density on the 2^{a+1} possible output words and, for different j's, these mappings are reasonably independent or orthogonal. One could use symmetric functions, modular arithmetics, or any of the conventional pseudorandom methods.*

Now, we think of the $b \cdot 2^{a+1}$-bit memory as organized into b-bit registers with $(a + 1)$-bit addresses: Suppose that $\mathbf{A}_{\text{file}}$ has already filed the words $w_1, \ldots, w_n$, and it is about to dispose of w_{n+1}.

A_{file}: Compute $R(w_{n+1}, 1)$. If the register with this address is *empty* put w_{n+1} in it. If that register is occupied do the same with $R(w_{n+1}, 2)$, $R(w_{n+1}, 3), \ldots$ until an unoccupied register $R(w_{n+1}, j)$ is found; file w_{n+1} therein.

A_{find}: Compute $R(w, 1)$. *If this register contains w, then* w is in the data set. If $R(w, 1)$ is empty, *then w is not in the data set.* If $R(w, 1)$ contains some other word not w, then do

*There is a superstition that $R(w, j)$ requires some magical property that can only be approximated. It is true that any particular R will be bad on *particular* data sets, but there is no problem at all when we consider average behavior on *all* possible data sets.

[illegible]

the same with $R(w, 2)$, and if necessary $R(w, 3)$, $R(w, 4)$, . . . , until either w or an empty register is discovered.

On the average, $\mathbf{A}_{\text{fin}}$ *will make less than 2b memory-bit references*! To see this, we note first that, on the average, this procedure leads to the inspection of just 2 registers! For half of the registers are empty, and the successive values of $R(w,j)$ for $j = 1,2, \ldots$ are independent (with respect to the ensemble of all data sets) so the mean number of registers inspected to find an empty register is 2.

Actually, the mean termination time is slightly *less*, because for w's in the data set the expected number of inspected registers is < 2. The procedure is useful for symbol-tables and the like, where one may want not only to know if w is there, but also to retrieve some other data associated (perhaps again by hash-coding) with it.

When the margin of redundancy is narrowed, for example, if

$$M = \frac{n}{n-1} \cdot b \cdot 2^a,$$

then only $(1/n\text{th})$ of the cells will be empty and one can expect to have to inspect about n registers.

Because people are accustomed to the fact that most computers are "word-oriented" and normally do inspect b bits at each memory cycle the following analysis has not (to our knowledge) been carried through to the case of 1-bit words. When we program $\mathbf{A}_{\text{find}}$ to match words *bit by bit* we find that, since half the words in memory are zero, matching can be speeded up by assigning a special "zero" bit to each word.

Assume, for the moment, that we have room for these 2^a extra bits. Now suppose that a certain w_0 is *not* in the data set. (This has probability $1\text{-}2^{a-b}$.) First inspect the "zero" bit associated with $R(w_0,1)$. This has probability $\frac{1}{2}$ of being zero. If it is not zero, then we match the bits of w_0 with those of the word associated with $R(w_0,1)$. These cannot all match (since w_0 isn't in the data set) and in fact the mismatch will be found in (an average of) $2 = 1 + \frac{1}{2} + \frac{1}{4} + \ldots$ references. Then the "zero" bit of $R(w_0,2)$ must be inspected, and the process repeats. Each repeat has probability $\frac{1}{2}$ and the process terminates when the "zero" bit of some $R(w_0,j) = 0$. The expected number of references can be counted

[illegible]

then as

$$\tfrac{1}{2}(1 + 2 + \tfrac{1}{2}(1 + 2 + \tfrac{1}{2}(\ldots))) + 1 = 3 + 1 = 4.$$

If w_0 *is* in the data set (an event whose probability is 2^{a-b}) and we repeat the analysis we get $4 + b$ references, because the process must terminate by matching all b bits of w_0.

The expected number of references, overall, is then

$$4(1 - 2^{a-b}) + (4 + b)2^{a-b} = 4 + b \cdot 2^{a-b} \sim 4$$

since normally 2^{a-b} will be quite tiny. We consider it quite remarkable that so little redundancy—a factor of two—yields this small number!

The estimates above are on the high side because in the case that w_0 is *in* the data set the "run length" through $R(w_0, j)$ will be shorter, by nearly a factor of 2, than chance would have it just because they were put there by A_{file}. On the other hand, we must pay for the extra "zero" bits we adjoined to M. If we have $M = 2b \cdot 2^a$ bits and make words of length $b + 1$ instead of b, then the memory becomes slightly *more* than half full: in fact, we must replace "4" in our result by something like $4[(b + 1)/(b - 1)]$. Perhaps these two effects will offset one another; we haven't made exact calculations, mainly because we are not sure that even this A_{file}-A_{find} pair is optimal.

It certainly seems suspicious that half the words in memory are simply empty! On the other hand, the best one could expect from further improving the algorithms is to replace 4 by 3 (or 2?), and this is not a large enough carrot to work hard for.

12.6.6 Summary of Exact Matching Algorithms

To summarize our results on Question 1 we have established upper bounds for the following cases: We believe that they are close to lower bounds also but, especially in cases 3 and 4, are not sure.

Case	Memory size	Bit-references	Method
2	$<(b - a)2^a$	∞	(impossible)
4	$(b - a)2^a$	$\tfrac{1}{2}(b - a)2^a$	(search all memory)
3	$b \cdot 2^a$	$\tfrac{1}{2}b \cdot a$	(logarithmic sort)
5	$2b \cdot 2^a$	$4 + \epsilon$	(hash coding)
1	$\geq 2^b$	1	(table look-up)

[illegible]

12.7 Time vs. Memory for Best Matching: An Open Problem

We have summarized our (limited) understanding of "Question 1" — the exact matching problem — by the little table in §12.6.6. If one "plots the curve" one is instantly struck by the effectiveness of small degrees of redundancy. We do not believe that this should be taken too seriously, for we suspect that when the problem is slightly changed the result may be quite different. We consider now

QUESTION 2: Given w, exhibit the word $\hat{w}$ closest to w in the data set.

The ground rules about $\mathbf{A}_{\text{file}}$ and $\mathbf{A}_{\text{find}}$ are the same, and *distance* can be chosen to be the usual metric, that is, the number of digits in which two words disagree. If $x_1, \ldots, x_b$ and $\hat{x}_1, \ldots, \hat{x}_b$ are the (binary) coordinates of points w and $\hat{w}$ then we define the *Hamming* distance to be

$$d(w, \hat{w}) = \sum_{i=1}^{b} |x_i - \hat{x}_i|.$$

One gets exactly the same situation with the usual *Cartesian* distance $C(w, \hat{w})$, because

$$[C(w, \hat{w})]^2 = \Sigma\, |x_i - \hat{x}_i|^2 = \Sigma\, |x_i - \hat{x}_i| = d(w, \hat{w})$$

so both $C(w, \hat{w})$ and $d(w, \hat{w})$ are minimized by the same $\hat{w}$.

12.7.1 Case 1: $M = 2^b \cdot b$.

$\mathbf{A}_{\text{file}}$ assigns for every possible word w a block of b bits that contain the appropriate bits of the correct $\hat{w}$.

$\mathbf{A}_{\text{find}}$ looks in the block for w and writes out $\hat{w}$. It uses b references, which is obviously the smallest possible number.

12.7.2 Case 2: $M < (b - a)\,2^a$.

Impossible, for same reason as in Question 1.

12.7.3 Case 3: $M = b \cdot 2^a$

No result known.

12.7.4 Case 4: $M = (b - a)\,2^a$

This presumably requires $(b - a) \cdot 2^a$ references, that is, all of memory, for the same reason as in *Question* 1.

[illegible]

12.7.5 Case 5: $(b - a)2^a < M < b \cdot 2^b$
No useful results known.

12.7.6 Gloomy Prospects for Best Matching Algorithms
The results in §12.6.6 showed that relatively small factors of redundancy in memory size yield very large increases in speed, for serial computations requiring the discovery of exact matches. Thus, there is no great advantage in using parallel computational mechanisms. In fact, as shown in §12.6.5, a memory-size factor of just 2 is enough to reduce the mean retrieval time to only slightly more than the best possible.

But, when we turn to the *best match* problem, all this seems to evaporate. In fact, we conjecture that even for the best possible $\mathbf{A}_{\text{file}}$-$\mathbf{A}_{\text{find}}$ pairs, the speed-up value of large memory redundancies is very small, and for large data sets with long word lengths there are no practical alternatives to large searches that inspect large parts of the memory.

We apologize for not having a more precise statement of the conjecture, or good suggestions for how to prove it, for we feel that this is a fundamentally important point in the theory of computation, especially in clarifying the distinction between serial and parallel concepts.

Our belief in this conjecture is based in part on experience in finding fallacies in schemes proposed for constructing fast best-matching file and retrieval algorithms. To illustrate this we discuss next the proposal most commonly advanced by students.

12.7.7 The Numerical-Order Scheme
This proposal is a natural attempt to extend the method of Case 3 (12.6.3) from exact match to best match. The scheme is

$\mathbf{A}_{\text{file}}$: store the words of the data set in numerical order.
$\mathbf{A}_{\text{find}}$: given a word w, find (by some procedure) those words whose first a bits agree most closely with the first a bits of w. How to do this isn't clear, but it occurs to one that (since this is the same problem on a smaller scale!) the procedure could be recursively defined. Then see how well the other bits of these words match with w. Next, . . . (?) . . . .

The intuitive idea is simple: the word $\hat{w}$ in the data set that is closest to w ought to show better-than-chance agreement in the

[illegible]

first a bits, so why not look first at words known to have this property. There are two disastrous bugs in this program:

1. When can one stop searching? What should we fill in where we wrote "Next ...(?)...." We know no nontrivial rule that *guarantees* getting the best match.

2. The intuitive concept, reasonable as it may appear, is not valid! It isn't even of much use for finding a *good* match, let alone finding the best match.

To elaborate on point 2, consider an example: let $a = 20$, $b = 10{,}000$. Let w, for simplicity, be the all-zero word. A typical word in the data set will have a mean of 5000 *one*'s, and 5000 *zero*'s. The standard deviation will be $\frac{1}{2}(10{,}000)^{1/2} = 50$. Thus, less than one word in $2^a = 2^{20}$ can be expected to have fewer than 4750 *one*'s. Hence, the closest word in the data set will (on the average) have at least this many *one*'s. That closest word will have (on the average) $> 20 \cdot (4750/10{,}000) = 9.5$ *one*'s among its first 20 bits! The probability that w will indeed have very few *one*'s in its first 20 bits is therefore extremely low, and the slight favorable bias obtained by inspecting *those* words first is quite utterly negligible in reducing the amount of inspection. Besides, objection 1 still remains.

The value of ordering the first few bits of the words is quite useless, then. Classifying words in this way amounts, in the n-dimensional geometry, to breaking up the space into "cylinders" which are not well shaped for finding nearby points. We have, therefore, tried various arrangements of spheres, but the same sorts of trouble appear (after more analysis). In the course of that analysis, we are led to suspect that there is a fundamental property of n-dimensional geometry that puts a very strong and discouraging limitation upon all such algorithms.

12.7.8 Why is Best Match so Different from Exact Match?

If our unproved conjecture is true, one might want at least an intuitive explanation for the difference we would get between §12.6.3 and 12.7.3. One way to look at it is to emphasize that, though the phrases "best match" and "exact match" sound similar to the ear, they really are very different. For in the case of exact match, *no* error is allowed, and this has the remarkable effect of *changing an n-dimensional problem into a one-dimensional problem!* For best matching we used the formula

$$\text{Error} = \sum_{i=1}^{b} |x_i - \hat{x}_i| = \sum_{i=1}^{b} 1\,|x_i - \hat{x}_i|,$$

[illegible]

where we have inserted the coefficient "1" to show that all errors, in different dimensions, are counted equally. But for exact match, since *no* error can be tolerated, we don't have to weight them equally: any positive weights will do! So for exact match we could just as well write

$$\text{Error} = \sum_{i=1}^{b} 2^i |x_i - \hat{x}_i| \quad \text{or even} \quad \text{Error} = \sum_{i=1}^{b} 2^i (x_i - \hat{x}_i)$$

because either of these can be zero only when all $x_i = \hat{x}_i$. (Shades of stratification.) But then we can (finally) rewrite the latter as

$$\text{Error} = (\Sigma\, 2^i x_i) - (\Sigma\, 2^i \hat{x}_i)$$

and we have mapped the n-dimensional vector $(x_1, \ldots, x_b)$ into a single point on a one-dimensional line. Thus these superficially similar problems have totally different mathematical personalities!

12.8 Incremental Computation

All the $\mathbf{A}_{\text{file}}$ algorithms mentioned have the following curiously local property. They can be described roughly as computing the stored information M as a function of a large data set:

$$M = \mathbf{A}_{\text{file}} \text{ (data set)}$$

Now one can imagine algorithms which would use a vast amount of temporary storage (that is, *much* more than M or much more than is needed to store the data set) in order to compute M. Our $\mathbf{A}_{\text{file}}$ algorithms do not. On the contrary, they do not even use significantly more memory capacity than is needed to hold their final output, M. They are even content to examine just one member of the data set at a time, with no control over which they will see next, and without any subterfuge of storing the data internally.

It seems to us that this is an interesting property of computation that deserves to be studied in its own right. It is striking how many apparently "global" properties of a data set can be computed "incrementally" in this sense. Rather than give formal definitions of these terms, we shall illustrate them by simple examples.

[illegible]

Example 1: We wish to compute the median of a set of a million distinct numbers which will be presented in a long, disordered list. The standard solution would be to build up in memory a copy of the entire set in numerical order. The median number can then be read off. This is *not* an incremental computation because the temporary memory capacity required is a million times as great as that required to store the final answer. Moreover it is easy to see that there is no incremental procedure if the data is presented only once.

The situation changes if the list is repeated as often as we wish. For then three registers are enough to find the smallest number on one pass through the list, the second smallest on a second pass, and so on. With an additional register big enough to count up to half the number N of items in the list, we can eventually find the median.

It might seem at first sight, however, that an incremental computation is precluded if the numbers are presented in a random sequence, for example by being drawn, with replacement, from a well-stirred urn. But a little thought will show that an even more profligate expenditure of time will handle this case incrementally provided we can assume (for example) that we know the number of numbers in the set and are prepared to state in advance an acceptable probability of error.

What functions of big "data sets" allow these drastic exchanges of time for storage space? Readers might find it amusing to consider that to compute the BEST PLANE (§12.2.3), given random presentation of samples, and bounds on the coefficients, requires only about three solution-sized memory spaces. One predicate we think cannot be computed without storage as large as the data set is:

[the numbers in the data set, concatenated in numerically increasing order, form a prime number].

In case anyone suspects that all functions are incrementally computable (in some sense) let him consider functions involving decisions about whether concatenations of members of the data set are halting tapes for Turing machines.

[illegible]