R. E. Bonner

A "Logical Pattern" Recognition Program

Abstract: A description is given of an IBM 7090 program which searches for "logical patterns" in a set of input samples. The program was tested on a character recognition problem, where it designed a recognition system whose error rate and hardware requirements are compared to those of a system designed by humans. This program is intended to be used as a research tool to discover certain kinds of patterns in data and as a step in the direction of automatic logic design for some character recognition problems.

Introduction

In recent years some attempts have been made to use a computer to generate properties useful in "concept" formation or pattern recognition. The results have been described by such authors as Bledsoe and Browning,[1] Hovland and Hunt,[2] Kochen,[3] Stearns,[4] and others. This paper relates another method of attacking the problem. The method has been implemented by three sequential IBM 7090 programs totaling about 8000 instructions.

The input to the first of these programs is a set of samples representing the problem. Each sample is a binary word with an identifying "name" or "type" attached to it. Each bit of the word represents the state of a certain property, present or absent. These properties are supposedly the result of a human designer's best effort to pick quantities pertinent to his recognition problem. After he has done this, if sufficient variation still remains among samples of a type, it is desirable to use some automatic technique to finish the design. How much effort the designer puts into finding the primary properties depends on how much power the automatic technique possesses and on the difficulty of the problem.

The output of the last program is a description of an automatically derived recognition system and the substitution error and reject rate for this system.

The basic theme of this method is twofold:

1) Use of nonexhaustive heuristic algorithms to find "valuable" properties (Program II).

2) Employment of various devices to make the requirements expected of heuristic method (1) less stringent. These include a method of dividing a problem into a number of simpler problems equivalent to the original problem (Program I) and a decision process which can operate effectively without requiring properties of very high "value" (Program III).

The practical rationale for attacking the problem at all rests mainly on these two points:

1) The pattern recognition problems of the future are apt to involve a tremendous latitude of variation in the patterns. Design by humans will be difficult because of the large amount of data that must be manipulated. Computers will have the ability to handle these data and can do so if we can tell them how.

2) Since automatic design produces a standardized structure, it permits concentration on the design program and the implementation of the type of machine the program designs. This is contrasted to the present typical situation involving a different design effort for each different problem.

Program I: Category formation

The first step taken to lighten the burden of the property-finding algorithm of Program II is to divide the original recognition problem into a number of simpler recognition problems which can be solved one at a time, independently of each other, and whose solutions taken together equal the solution to the original problem. The program accomplishes this by dividing the original sample set (S) into a number of mutually exclusive sample sets (P_i) and insuring that any new sample will be placed in one and only one of these sets. Each of the P_i, henceforth called a *category*, can now be looked at as representative of a recognition problem in itself and considered separately. This process is depicted in Fig. 1 for five categories.

The criterion for categorizing requires mutual "similarity" among members of a category. Since the most bothersome recognition problems occur when samples which look alike have different "names," this choice of criterion tends to isolate and simplify the difficult parts of the original problem. For example, in alphabetic character recognition, many samples of *O* and *Q* might "look alike" and tend to be placed in the same category, while presumably most of the other letters would be sufficiently different to be excluded from this category. If they were alone in that category, the required recognition logic would involve a simple search for the tail of the *Q*.

Of course, it is entirely possible that not all samples of a "type" look alike (i.e., *A* and *a*) and so different samples of the same letter can be present in a number of categories.

During the categorizing process, the "name"(such as *A*) of a sample is not referred to, in order that the program be unprejudiced by human preconception regarding what "types" should look alike. The reliance is solely upon a measure of "similarity" between two samples. It is possible to define and use many measures of similarity, each pertinent to its own problem. However, in order that specific examples of the categorizing process can be given, a particular definition of similarity will be used throughout:

To compare two binary words, count the number of identical bits in the words and compare this to a present number T. If it equals or exceeds that number, the words are similar.

This definition was chosen because:

1) There is a one-to-one correspondence between "difference" and distance in *n*-dimensional property space.

2) In the absence of other knowledge, "0" and "1" are two states which should be treated identically.

3) It provides simplicity and ease of computer manipulation.

• *Method of formation*

The categories are formed in the following manner:

a) Take Sample 1; remember it as "Mask 1".

b) Take Sample 2 and compare it with Sample 1 to see if they are similar.

c) If they are similar, ignore Sample 2; if they are not, remember Sample 2 also as "Mask 2".

d) Take Sample 3 and compare it, one at a time, to all samples which are being remembered as "masks".

e) If it is similar to any of these, ignore it; if not, remember it as another mask.

f) Continue this process until all samples have been used up. The result will be a number of masks, none of which are similar to any other. For an example, see Fig. 2.

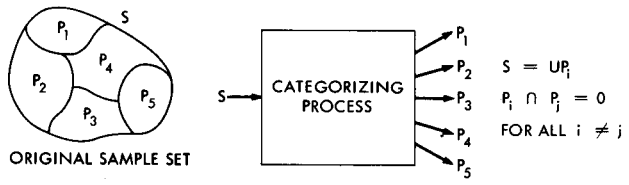


Figure 1 **The categorizing process.**
The original recognition problem (sample set S) is divided into several simpler recognition problems P_i.

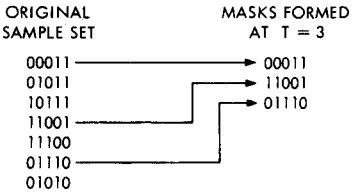


Figure 2 **Method of forming the "masks".**

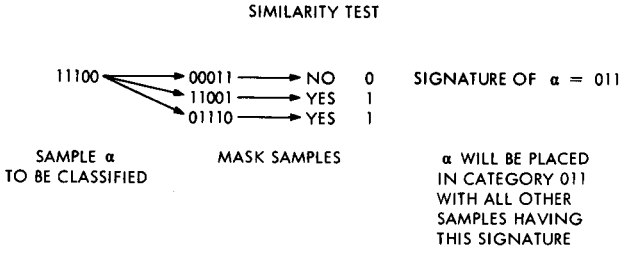


Figure 3 **Example of "signature" formation using the program's definition of similarity.**

g) Return to the first sample of *S* and compare it in turn to each of the mask samples. A binary word representing the result of these comparisons is formed, with one bit per comparison. If the sample is similar to a mask sample, then the bit representing this mask sample is a ONE; if not, it is a ZERO. Call the binary word formed for a sample in this manner the *signature* of the sample. All samples with the same signature are placed into the same category and that signature becomes the "name" of the category.

The program therefore assigns a signature to each sample, determines the number of different signatures, and groups together all samples with the same signature. The process of signature formation is illustrated by Fig. 3.

This method of finding masks and signatures repre-

sents a compromise between sophistication and practical realization. Since it is a pre-processing step, it was decided to lean heavily toward producing a fast program. The masks found represent a set of pseudo-orthogonal vectors used to characterize the other samples. If there are n masks, not all 2^n possible categories need actually be present. In fact, since there is a guarantee that each sample is similar to at least one mask, the category represented by an all-ZERO signature is never present. All categories not used by the original sample set represent the "reject" space. For each category used, a logic is designed by the following programs using the samples in that category. When the process is completed, the resulting recognition system will examine an unknown sample (assumed to be not in the original set) and determine its category by finding its signature, after which it will either reject it or proceed to use the recognition logic appropriate to this category.

A simpler way to determine masks would be to pick uniformly scattered points as category "centers". The advantage of the present method is that only that part of measurement space containing samples is selected to be dissected. The simpler method considers the whole space and would produce inefficient results in situations where all samples were in a relatively small portion of the space.

A disadvantage of the present method is that, other than guaranteeing consideration of the correct area in the space, it disregards any natural clustering boundaries. For instance, it may divide a cluster of A's in half, even though the members of the cluster are all similar to each other. This results in relatively inefficient masks as compared to those which could be obtained from a more sophisticated method which took natural clustering into consideration.

• *Program control*

1) In running the program, the user specifies the number of masks he desires, say m, and the program manipulates the threshold T, automatically attempting to satisfy this request. Since the number of signatures or categories can never be larger than $2^m - 1$, this feature provides a control on the maximum number of sample sets formed. The number desired is left to the judgment of the user and normally depends on the number of possible types of samples and the difficulty of the problem.

2) There is an option to subcategorize within each category. The threshold of similarity finally arrived at in the original categorization process is used to look for a set of samples (new masks) within a category which are all dissimilar at that threshold. It is sometimes possible to find such a set because members of a category are not guaranteed to be similar to each other at the categorizing threshold (only at worst at twice this threshold). The normal process of categorization is followed, which ultimately results in a number of new categories for those of the former categories which can be subcategorized in this manner.

3) There is a feature which allows the program to ignore samples of a certain type in a category after it has more than a certain preset number. This is valuable in cases where two types have a large difference in frequency of occurrence in a category. For example, if there are 1000 A's for each B, then to get a reasonable number of B's to examine, one would have to examine many more A's than were needed to generalize. The feature allows one to stop collecting A's after a certain number, say 200, has been reached, while continuing to collect all the B's available. Memory requirements of Program II also dictate the need for such a feature.

4) There is a mode of operation which allows the masks formed from one sample set to be used in forming signatures for samples of a different set. This allows testing of a recognition system on samples not used in its formation to determine error-rate stability.

Some of the advantages of this categorization scheme are:

1) It allows separate consideration of the recognition problems in each category. An example of the use of this property is given in the description of the results.

2) It allows simple control of the tradeoff between error rate and equipment cost. By specifying a large number of masks, the user can most likely get an overall decrease in error rate but will require more hardware to effect it.

In summary, the input to Program I is a set of samples. The output is a number of subsets of this set, where the members of each subset are "similar" to each other.

Program II: Property-finding algorithm

This section considers each category from Program I separately and treats each in the same manner. Essentially, it looks for bits and logical combinations of bits which are valuable in distinguishing the various types of samples in the category. It does this mostly in a nonexhaustive manner by proposing hypotheses of possibly valuable properties, then evaluating and classifying these hypotheses and creating new hypotheses from the result. The following description of the process operates on the samples within a particular category.

• *Definititions of "high" and "good"*

The measure of "value" of x_j (the j^{th} state of property x) relative to type of sample A_i is defined by use of $P_{A_i}(x_j)$, the conditional probability of x_j, given that the sample is an A_i. This can be estimated simply by counting the number of samples of A_i for which x_j is present, defined as $N_{A_i}(x_j)$, and dividing by the total number of A_i samples, defined as N_{A^i}.

There are by definition, two components of value:

1) x_j is "high" with respect to A_i if

$$P_{A_i}(x_j) > K_1 ,$$

where $1 \geqq K_1 \geqq 0.5$.

2) x_j is "good for A_i vs A_k" if both equations (a) and b) below are satisfied:

$$\frac{P_{A_i}(x_j)}{P_{A_k}(x_j)} > K_2 \qquad \text{(a)}$$

$$\frac{N_{A_i}(x_j)}{N_{A_k}(x_j)} > K_2 \qquad k \neq i, \quad K_2 > 1 . \qquad \text{(b)}$$

The parameters K_1 and K_2 are arbitrarily picked by the designer and typical values might be $K_1 = 0.9$, $K_2 = 4$. Since the program tests for presence or absence of a property, x_j is always either ONE or ZERO.

The presence of "highness" guarantees that a particular state of a property is present quite often for a particular type of sample. This, however, is not enough to make it valuable, for it may be present quite often for the other types of samples also. The presence of component (b) of "goodness" guarantees that this is not so. Component (a) of "goodness" insures discriminating power. Table 1 gives some examples to familiarize the reader with the two "goodness" components.

A state of a property relative to A_i may be good for some of the A_k and not for others. It is called "good for all" if it is good for all A_k ($k \neq i$).

Table 1 **Examples of violation of "goodness" for $K_1 = 0.9$, $K_2 = 4$.**

	1	2	3	4
N_{A_i}	100	10	100	100
N_{A_k}	10	100	100	100
$N_{A_i}(x_j)$	91	10	91	100
$N_{A_k}(x_j)$	10	10	10	100
Eq. (a) satisfied	No	Yes	Yes	No
Eq. (b) satisfied	Yes	No	Yes	No

The measures of value chosen are relatively simple and have the advantage of indicating what type of value a property possesses, thereby allowing more intelligent use to be made of this property. For example, knowing that a property is valuable for A vs B allows it to be used in resolving A-B conflicts. Also, the method of combining properties to produce more valuable properties is based on use of value type to indicate the course of action.

• *Summary of property combination procedure*

It is clear that there are some possible classifications of a state of a property (henceforth called merely *property*), which can be derived from "high" and "good" such as:

(1) High and good for all (HGA)

(2) High and good for some (HGS)

(3) Good for all but not high (GA)

(4) Good for some but not high (GS).

The problem is to take properties which have classifications (2) to (4) and convert them to properties of class (1), the most valuable. Because of its general nature, logical combination seemed a fitting method.

For instance, OR-ing two properties together will produce a property which occurs at least as frequently (i.e., with at least as much "highness"), as either of the two original properties. Therefore, OR-ing two properties classified GA might yield one classified HGA. The logical AND on the other hand, lowers the "highness" and can help to produce "goodness" if it lowers the "highness" of competing types more than that of the desired type. By a procedure using these principles, the program builds up properties of such complexity as: a thirty-way AND OR-ed to a forty-way AND; a four-way OR AND-ed to another; and other properties of lesser complexity down to and including individual bits.

The following is a detailed description of this procedure, which is also represented in Fig. 4.

• *Property search: Part I*

First, the $P_A(x)$ and $N_A(x)$ are calculated for both states of each bit for all bits and for all types. The types are now considered one at a time.

The two states (b_r and $\bar{b}_r$) of each bit position r are classified for the first type, say A_i, as having certain "kinds" of value (i.e., HGS, GA, et cetera).

The bit states found to be HGS are remembered simply as valuable for A_i. They are also stored in a pair "conflict" memory where they are remembered as valuable for A_i vs A_k. Any HGS properties subsequently found are also stored in the "conflict" memory.

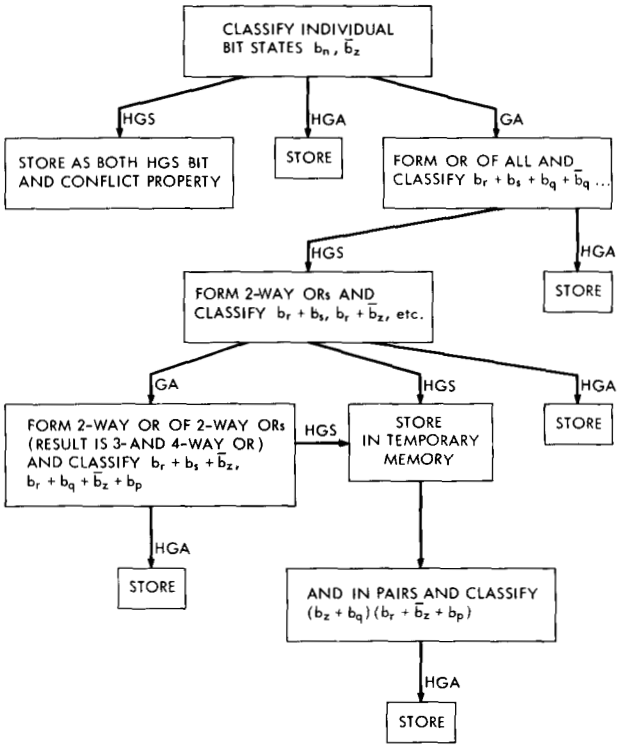
The bit states found to be HGA are remembered as exceptionally valuable for A_i. Any properties subsequently found to be HGA are remembered in the same way. Therefore, this classification will not be mentioned again.

All the bit states found to be GA are used to generate a many-way OR (example, $b_r + \bar{b}_s + \bar{b}_z + b_q \cdots$). The presence of this property is now tested for in the sample set and the property is classified.

If it is GA, it is ignored since "highness" has not been produced.

If it is HGS, it means that too many bits have been used in the OR and GA-ness has been lost. In this case, all possible two-state OR's of GA bit states (example, $b_r + \bar{b}_z$) are formed to be tested. It can be seen then that this many-way OR acts as a quick check to tell if OR-ing any subset of GA bit states will ever produce "highness".

Part I



Part II

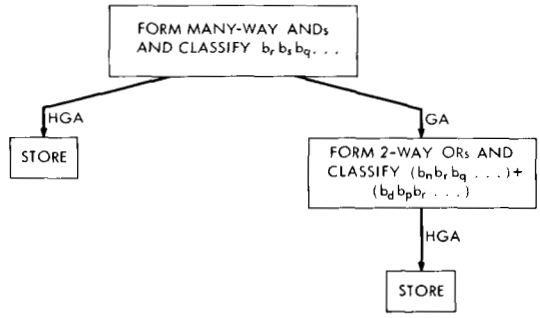


Figure 4 **Procedure of searching for properties.** *The contents of the boxes is a description of the synthesis performed and a sample result. The arrows from each box indicate the subsequent path for properties so classified.*

The two-way OR's are classified. If an OR is HGS, it is stored in a temporary memory (not the conflict memory) for future use. All OR's found to be GA are again OR-ed in pairs exhaustively producing three- and four-way OR's (i.e., if $b_r + \bar{b}_z$ and $\bar{b}_z + \bar{b}_s$ are both GA, then the new OR is $b_r + \bar{b}_z + \bar{b}_s$).

These new OR's are classified. All OR's which are HGS are stored in the same temporary memory used for HGS two-way OR storage. All possible two-way AND's are now formed from the properties in this memory. Excluded are two-way AND's where one component is a subset of the other because this property has already been tested

$$\{\text{i.e., } [(b_r + \bar{b}_z)(b_r + \bar{b}_z + \bar{b}_s) = (b_r + \bar{b}_z)]\}.$$

The hope is that this AND-ing will restore some lost "goodness". These AND's are classified and any result other than HGA is ignored.

• *Property search: Part II*

The next section of the program (Part II in Fig. 4), generates many-way AND's (sample, $b_r b_s \bar{b}_q b_z \cdots$). It does this by AND-ing together the first two samples of the same type and then counting the number of ONE's in the resultant AND. If this number is above a certain preset amount, another sample of this type is AND-ed in; if it is not, the last AND is stored and the sample which caused the bit sum to drop below the acceptable limit is used to start a new AND. This process is continued until all samples of the type are used.

The idea here is to generate many-way AND's which are present often for the type from which they were generated but, because a large number of conjunctions can be required, are restrictive enough not to be found much in other types.

These hypotheses are evaluated. Those classified GA are OR-ed in pairs $(b_r b_s b_q \cdots + b_r b_z \bar{b}_n \cdots)$ exhaustively and evaluated, in the hope of generating highness. These AND's are classified and any result other than HGA is ignored. The memory can accommodate five properties per conflict per category and sixteen HGA properties per type per category.

This ends the search for properties for type A_i. The same search method is now used for all other types in the category.

• *Additional features*

In addition to the search for properties, a test is made after the individual bits have been found to see if enough are available to identify the types. If there are, the program stops looking for properties. This is done to avoid needless complication.

Statistics are compiled on the number of each kind of property that is found. If not enough properties are found, the program lowers its criteria of "value" to an alternate set of K_1 and K_2 also specified by the designer and goes through the whole procedure again.

When Program II is finished, it has for each type in each category:

1) A set of HGA properties.

2) A set of HGS bits.

3) A set of "conflict" properties for that type vs each of the other types in its category. A typical one might be classified as "high and good for A vs B" and is used by Program III to resolve A-B conflicts.

Program III: Recognition criteria

This program implements the decision-making operation which attempts to recognize the given samples using the properties found by Program II. Here again each category is examined separately, its samples being recognized by the properties found for that category.

The decision-making philosophy evolved from the consideration that it should be useful even if Program II finds no HGA properties. It also attempts to make efficient use of the knowledge which the value criteria of Program II provide. This is best illustrated in the use of the "conflict" set of properties.

• *Decision procedure*

A sample is examined for the presence of each of the HGA and HGS properties of each type. A percentage match is obtained for each type. This is obtained by finding the number of properties present and dividing by a normalizing factor. The highest percentage match is determined and the percentage match for each of the competing types is subtracted from it, one by one. Each of those types whose match is within 0.3 of the highest are considered part of the conflict group. If none are within that amount, the sample is identified as the type of the highest match.

If there is a conflict group, the conflict properties are used to resolve it. Each conflict property is assigned a weight, starting at a value of "one". Each set of properties for a particular conflict has a normalizing factor which is initially equal to the number of properties present.

As an example of the process followed when a conflict appears, consider that an unknown sample produces a conflict group of A, B, and C. This involves six sets of conflict properties; namely A vs B, A vs C, B vs A, B vs C, C vs A, and C vs B. The sample is examined for the presence of all of these properties. Let us assume that the A vs B property set consists of three properties with weights W_1, W_2 and W_3 and normalizing factor N_{AB}. Let us assume also that only properties (1) and (2) are present. The percentage match for A vs B in this case is then given by $W_1 + W_2$ divided by N_{AB}. In like manner, the percentage match for A vs C is computed and averaged with that for A vs B to yield an overall percentage match to represent the type A in this conflict situation. The conflict percentage match for A is then averaged with the original percentage match to obtain the final percentage match for A. Identical computations are performed for B and C.

The sample is finally identified to be the type of the highest match. There is also a reject feature which allows the sample to be rejected if the closest competitor to the highest match is within a certain amount.

• *Perturbation learning*

As a further feature, learning can occur if a reject or error is found. This might be described as *perturbation learning*, since its aim is to compensate for the "odd" samples which the "bulk" statistical learning process of Program II cannot handle. This learning process attempts to alter the recognition system until the error or reject is removed or until the learning feature gives up. It operates in two ways:

1) It can alter the multiple AND properties.

2) It can reassign weights to the conflict properties.

When an error or reject is found in a conflict situation, the program alters the weights of the conflict properties in the following manner:

1) If the property in question was in the proper state (present for the right type or absent for other types), its weight is increased by one, as is the normalizing factor for its conflict set.

2) If the property in question was in the wrong state (absent for the right type or present for a wrong type), its weight is decreased by one, as is the normalizing factor for its conflict set. If the weight is already zero, nothing is done.

Only those properties involving conflict with the correct type are altered. For example, if the proper type is A, properties in (B vs C) are not altered, even if both are in the conflict set.

This choice of learning procedure allows repeated learning from the same sample to produce conflict percentage matches which usually approach 100% for its actual identity and something less for all other conflict types involved.

The second learning feature involves changing the many-way AND properties in the high-and-good-for-all class which were not present for the correct identity. The AND property is AND-ed to the sample in error and the ONE's in the resultant word are counted. If this sum is above the preset threshold, the old AND is replaced by this new AND. The philosophy here is that a 35-way AND is almost as restrictive as a 40-way AND and therefore, if the number of ONE's in the AND is high, chances are it will still retain its goodness. Its highness will increase and it will now be present for the error sample where it was previously absent. In any case, because it retains its highness, the worst effect of such a move would be to create more conflict situations. If the correct identity was one of the alternatives before learning was effected, it would still be an alternative.

In addition, the program keeps testing and learning until it removes all errors or until its error rate remains constant (or rises).

Results of program test

The input set in the test consisted of 27,519 samples of E13B type font numerals (10) and special characters (4). No two samples were completely identical. This set represented the result of an exhaustive search through about 1,000,000 electronically scanned character samples to obtain a statistical cross section of what the machine will have to read. It was the same set that was used by human designers in obtaining the logic for an IBM character recognition machine. The samples were represented by a 7 × 10 bit binary matrix. This bit matrix provided the standard quantized visual form of the character. The E13B type font is very stylized and designed for ease of machine recognition. However, the numbers of different samples generated and the low error rate required still rate it as a nontrivial test of a machine process.

The goal was to use the program to design a recognition system for these samples and compare its performance against that of the human designers. The same set of 27,519 samples used in the design were employed to test this design.

• *Specific design details*

In using the program, the aim was to obtain a satisfactory error rate at as low a cost as possible.

In the first attempt, 90 categories were used, and the results were that all samples were recognized correctly by the recognition system found. There were no decisions where the top choice had a competitor whose percentage match was within 0.15 (perfect score is 1.0). Since this was used as a reject criterion, there were no rejects.

The number of categories was then lowered to 30 in an attempt to reduce cost. Two of these categories had more than 3000 members. This created a problem because the program was restricted to handling a maximum of 3000 samples per category in forming the recognition system. There was an overflow of 5629 characters. However, Part I of the program has an option to save overflow and so it did. The idea here is that the perturbation learning feature can later be used to adjust the initial system to conform also to the overflow samples.

In the initial run there were no substitution errors and 7 rejects in 21,890 samples. After adding the overflow for one pass at perturbation learning, the over-all error rate was three substitution errors and 20 rejects. While it was realized that this could probably be improved by another pass through perturbation learning, an attempt was made to bypass this step by trying a smaller number of categories.

Fifteen categories were tried. Here there were three categories overflowing, containing a total of about 12,000 samples. About 9000 of these were in one category. This is rather more than a perturbation, so perturbation learning was not attempted here. In addition, the 15-category system did not offer a cost saving over the 30-category system (because of the increased logical complexity required per category).

The next step here was to let the computer design a new 30-category system with tightened controls (more "value" required of a property before it was retained) to produce a less expensive model. The result was a system having about one-fourth the number of components of the previous system. However, when tested for error rate, one category in this system produced 1036 rejects out of 1942 samples, whereas it previously had two rejects. Because of the independence of categories, it is possible to use the design for this category found in the previous 30-category system in place of the new design for this category and to use the new design for the other 29 categories. This combination yielded an error rate (without considering excess) of no substitution errors and two rejects out of 21,890 samples.

The overflow was tested with the new system for the two categories involved. After one pass through perturbation learning, one category had no substitution errors and three rejects and the other had no substitution errors and 112 rejects. Therefore, one of the new categories was accepted and the other deemed unsatisfactory.

At this point it was decided that enough was done to illustrate the procedure, although it seemed likely that further manipulation could cut the system cost and also improve performance.

• *Comparison with human design*

The final system had 30 categories (two old and 28 new). It made two substitution errors and had 14 rejects out of 27,519 samples. There were about 4000 individual bit tests, 110 many-way AND's, 7 OR's and 66 disjunctions of AND's. About 10 hours of IBM 7090 time were required for the design.

Assuming a particular hardware configuration, it is estimated that this system would have about three times the number of transistors as the human-designed machine logic system. This must be balanced against the monetary saving from a less expensive design effort and the savings in time to produce a product. This estimate of the size of the machine-designed system is based on hardware which would be useful only for the problem which generated it and would be incapable of further learning.

The speeds of the two machines would be comparable.

The real error rates could not be compared exactly, because the frequencies of occurrence of the samples in error and rejected in the experiment were not known. Also, it is not known how the design would do on another set of new samples. However, under the assumption that the million samples used in the design were statistically sufficient and because of the small number of errors obtained and the likelihood that they had lower than average frequencies of occurrence, it is not unreasonable to guess that the error rates of the

two machines would be similar. Although it was certainly desirable to explore this matter further, the unfavorable cost differential between machines made it somewhat premature to do so. Consequently, present effort is being directed toward improving the program.

Conclusions and discussion

The over-all method presented points up the usefulness of the concepts of "similarity" of samples and "value" classification of a property as tools to reduce the exhaustive search problem in automatic design. Specific points are:

1) The method of finding conjunctive "concepts" is different from those of Kochen or Stearns in that it does not use samples of negative instances but relies on the oddness of the many-way AND to provide goodness.

There seems to be some philosophical import to this difference. The many-way AND provides a "blanket" goodness. The more ONE's required by the AND, the smaller the universe of samples which can satisfy it. Therefore, if such an AND can be found present in most samples of a type, it can act as an effective filter rejecting *everything* except things very "similar" to the samples which generated it. When the AND is lessened to a few bits, this blanket rejection is lost, being sacrificed for efficiency. Therefore, for problems where the negative-instance set is unmanageable, it could prove to be worth a sacrifice in efficiency if this rejection could be better retained.

2) Using a normal logic-reducing program, it might be impossible to generate a sufficiently small number of members of a disjunction of many-way AND's to be practical. However, since the categorizing process of Part I separates the samples of a type into groups where the members are similar, it is easier to find many-way AND's in each category. This emphasizes the role of similarity between samples in leading to a more efficient answer (since fewer disjunctions are required). Rather than indiscriminately AND-ing together samples, if they were first broken into a number of "clumps" of look-alike samples, and each clump searched separately for AND's, it would seem that more independence between AND's would be obtained, and therefore probably fewer AND's would be required.

3) The procedure used to find valuable properties used the sample set representing the recognition problem to generate them, rather than some random-choice technique. Although this is not new, it is felt that a great gain in efficiency is realized by doing this.

As a final summary, a comparison was made between machine and human performance in a practical situation. The results seem to indicate that efforts in automatic recognition logic design are feasible and, in all probability, can ultimately yield some useful solutions. Since the program presented has thus far been tested only on one task which humans did not find too difficult, it remains to be seen how large a class of problems it and its successors can handle successfully.

Acknowledgment

I wish to thank H. Penafiel for programming Part I.

References

1. W. W. Bledsoe and I. Browning, "Pattern Recognition and Reading by Machine," *Proceedings of the Eastern Joint Computer Conference*, 225 (1959).
2. E. B. Hunt and C. I. Hovland, "Programming a Model of Human Concept Formation," *Proceedings of the Western Joint Computer Conference*, 145 (1961).
3. M. Kochen, "An Experimental Program for the Selection of Disjunctive 'Hypotheses'," *Proceedings of the Western Joint Computer Conference* **19**, 571 (1961).
4. S. D. Stearns, "A Method for the Design of Pattern Recognition Logic," *IRE Transactions on Electronic Computers*, **EC-9**, 48 (March 1960).

Received June 19, 1961