Experiments in the recognition of hand-printed text: Part II—Context analysis

by RICHARD O. DUDA and PETER E. HART

Stanford Research Institute
Menlo Park, California

INTRODUCTION AND BACKGROUND

Problem specification

The work described in this paper is part of a larger effort aimed at the recognition of hand-printed text. In a companion paper, Munson[1] describes the scanning of the text, and the preprocessing and tentative classification of individual characters. In this paper, we describe techniques for using context to detect and correct errors in classification.

The source text used in this experimental study consisted of hand-printed FORTRAN programs. The choice of this common programming language gave us a problem in which contextual relations were both fairly elaborate and well defined. This is to be contrasted with simpler problems, such as might be encountered with business forms, where contextual relations are rudimentary, and with the much more difficult problem of handling natural language, where complex semantic considerations play a large role.

The techniques we developed are embodied in a LISP program called the *context-directed analyzer*. The input to this program, which is obtained from the pattern classifier, consists of a list of possible alternative classifications for each character in the source program. Associated with each alternative is a number that measures the confidence that the alternative is in fact correct. Thus, if presented with a hand-printed A, the classifier might produce the output

Choice	Character	Confidence
1	R	−22
2	A	−28
3	H	−43

indicating an erroneous first choice, but a correct second choice ranking near the first in confidence.*

*The way in which these confidences are actually obtained is described in a later section. Ideally, each confidence is proportional to the logarithm of the probability that the corresponding alternative is correct.

The input to the context-directed analyzer is a list of such lists of alternatives and confidences for all of the characters in a FORTRAN program which we assume to be syntactically legal. The task of the analyzer is to achieve as low an overall error rate as possible by making appropriate choices from among the alternatives.

Past approaches

The utilization of contextual constraints to improve the performance of pattern classifiers has been the subject of a number of investigations.[2–11] One of two basic approaches has generally been followed, the table look-up method or the Markov approach. The table look-up method is based on the assumption that every word in the text is selected from a known finite table. A word of text is classified by comparing it with every table word having the same length and finding the best match. Gold[2] used such a table of legal Morse-code symbols in his system for recognizing hand-sent Morse code, and Bledsoe and Browning[3] used a table of English words in their pioneering experiments in recognizing hand-printed characters.

The Markov approach is rooted in the assumption that the true category of a character is related in a probabilistic manner to the true categories of a small number of surrounding characters. Its use leads to the estimation, from sample text, of the probabilities of all possible pairs, triples, or in general n-tuples of characters. The Markov method can be expected to correct locally improbable character strings, but it ignores global considerations. Harmon[4] used this method to detect errors in the recognition of cursive script. Edwards and Chambers[5] and Carlson[6] also employed this technique to correct errors encountered in conventional optical character recognition. An interesting mixture of the two approaches was used by McElwain and Evens[7] to correct garbled Morse code, and both methods were compared experimentally in a lucid paper by Vossler and Branston.[8]

Both of these approaches rest on the theoretical

foundations of compound decision theory (see Abend [9] for a clear tutorial presentation and for references to the appropriate statistical literature). Both Abend[10] and Raviv[11] point out the importance of considering the alternatives that can be supplied by a classifier. They derive the formal decision-theoretic solution for the optimum use of context and show how it can be simplified by the Markov dependence assumption. However, this assumption, which seems necessary to make the optimum procedure computationally feasible, again limits the ability to exploit global relations.

A formal solution

In this section we outline the general formal solution to the compound decision problem and then point out its drawbacks.

The solution

Suppose for a moment that we have on hand the output of the classifier for a single FORTRAN statement. The basic problem is to select, from among all the alternatives, the correct string of characters. A formal solution to this problem is provided by compound decision theory and requires two ingredients. We must be able to specify, for an arbitrary string of characters (1) the confidence of the string, and (2) the prior probability of the string. The former is provided by the classifier, while the latter must be assumed or estimated ahead of time. A formal solution, derived and made precise in the Appendix, is given by the following intuitively appealing rule:

> Compute the confidence of every string of characters of the given length. Bias each string confidence by adding the logarithm of the prior probability of that string. Set the answer equal to the string having the highest biased confidence.

If we assume that the confidences of the individual characters are the logarithms of their probabilities, then the confidence of a string is just the sum of the confidences of each of the characters in the string. We shall adopt the convention that any character not explicitly listed by the classifier as an alternative is correct with some uniformly low probability, and hence has an accompanying confidence of some low number.

Let us illustrate this rule with the following example. Suppose the classifier returns, for a single FORTRAN statement, the following alternatives, where for readability we list all first choices on the first row, second choice on the second row, and so on.

6	O	T	D	S
G			O	5
			=	

We display the associated confidence of each alternative in a similar array:

−60	−20	−30	−50	−30
−70			−60	−40
			−70	

In order to use the formal compound decision theory solution, we must know the prior probabilities. Let us assume initially that all legal strings of characters are equally likely, and also that all illegal strings have zero probability. The highest confidence string is 6OTD S, with confidence -190, but has prior probability zero. In fact, of the twelve possible strings of characters that can be formed from the alternatives presented, all but four are illegal FORTRAN statements. The four legal ones are

String	Confidence
GOTO S	−210
GOTO 5	−220
GOT = S	−220
GOT = 5	−230

Therefore the final selection is the assigned GO TO statement GOTO S.

In this example we have been casual about the existence of spaces in the text, and throughout the paper we shall ignore questions of spaces and other pragmatics (continuation marks, separating the label field from the rest of the statement and the like) since character position information is assumed to be provided by the original data scanning and input routine.

The disadvantages of the formal solution

The formal solution illustrated above suffers from at least two serious problems: A combinatorial explosion, and an inability to exploit semantic information in a natural way.

Problem 1: Combinatorial explosion

The first objection to the formal solution is that it rapidly gets out of hand combinatorially. Notice that the solution requires explicit enumeration of all possible strings of characters of the given length. We can, of course, adopt the reasonable heuristic that strings of characters will be formed only from among alternatives specifically listed by the classifier. Even so, a statement only ten characters long with, say, four alternatives for each character gives rise to over a million possibilities.

Of course, if we could order strings by confidence then we need not enumerate all possible strings since the decision procedure outlined above selects the highest con-

fidence legal string as the final answer. Thus we could examine the highest confidence string, and if it were legal we would have the answer. If not, we could examine the second most confident string, and so on. In general, this approach requires a method for selecting the k^{th} most confident string of characters given the $(k-1^{st})$ most confident. The problem of ordering strings by confidence is far from trivial. A solution, based on a modification of dynamic programming as suggested to us by R.E. Larson of Stanford Research Institute, is described in Reference 12. While the dynamic programming approach is considerably more efficient than the brute force approach and is used frequently in the analyzer implemented, it also suffers from severe combinatorial problems and can be used only on combinatorially simple data structures. For our computing facilities, the limit of combinatorial complexity for dynamic programming seems to be something on the order of a few thousand combinations; i.e., a string of five or six characters, with about four alternatives for each one.

Problem 2: Semantics

The decision rule discussed above involves only the syntax of the FORTRAN language. It by passes all the richness of semantics. For example, it ignores the simple but important fact that identifiers, and especially variable names, rarely appear only once in a given program. In principal, compound decision theory does not ignore this; it is taken into account by the prior probability of an entire program. Thus, for example, a program containing the variable name HELLO only once is less probable, a priori, than a very similar program containing HELLO several times. In practice, certainly, it is a hopeless task to reflect the multiple appearance of an identifier by directly enumerating prior probabilities.

Structure of the contex-directed analyzer

The previous section outlines the decision-theoretic approach to the utilization of context and points out two severe drawbacks. This section describes the structure of a context-directed analyzer that retains the flavor of the decision-theoretic solution while minimizing combinatorial problems. The analyzer capitalizes on the multiple appearance of variable names, which is a first step toward employing semantic, in addition to syntactic, information to correct classifier errors. We briefly describe the overall operation of the analyzer and then describe its major operations in more detail.

Let us first delineate the current status of our work. The analyzer described is implemented as a LISP program running on an SDS-940 computer. The data for the classifier is an SDS FORTRAN II program which is restricted only in that the I/O lists in input-output statements are simple lists of identifiers. Every statement in the FORTRAN program, except for the COMMENT and FORMAT statements, is subjected to a detailed analysis. For these particular statements the analyzer as yet returns the first choice for each character as the answer.

The analyzer is organized as a two-pass program. During the first pass each statement is identified by type and a table of identifier names is assembled and clustered. During the second pass each statement is resolved and the final classification of the FORTRAN text is made.

In the subsequent discussion we will need to refer to the data structure, illustrated in the GOTO example, that the classifier produces when presented with a segment of FORTRAN text. We will, for no special reason, refer to this data structure as a *P-list*. The i^{th} element of a P-list is a collection of alternatives and confidences for for the i^{th} character in the original text. Thus a P-list contains all the information passed from classifier to analyzer, and the elements in the P-list are in one-to-one correspondence with the characters in the original segment of text.

Statement identification

The identification of statement type is enormously facilitated by the appearance of a "control word" (DIMENSION, IF, etc.) at the beginning of all statements except the arithmetic assignment statement. This property of FORTRAN gives us good reason to suppose that our early assumption that all legal strings are equally likely grossly oversimplifies the matter, at least when the string is at the beginning of a statement. A more realistic assumption would be that each of the 30-odd FORTRAN control words is equally likely at the beginning of a statement, but this would overlook both the relative frequencies of statement types and the arithmetic assignment statement. A simple and not too unrealistic course of action would be to treat all control words as being equally likely, but to reject them all in favor of the arithmetic assignment statement if a sufficiently good match with a control word is not found. This leads directly to the following procedure for identifying statement type.

> Compute the confidence of each FORTRAN control word from the leading segment of the P-list for the statement. If the highest confidence computed exceeds a threshold, then decide the statement is of the corresponding type. Otherwise, decide the statement is an arithmetic assignment.

At this point we must make a remark about the computation of the confidence of a string of characters. If

each confidence were in fact the log of a probability, then the confidence of a string would be (under a conditional independence assumption described in the Appendix) the sum of the confidences of each component in the string. This is, unfortunately, not the case in practice. A reasonable measure of the confidence of a string that has proven quite satisfactory in practice is the normalized, or average, confidence, i.e., the sum of the confidences divided by the length of the string being considered. As an illustration, the confidence of GOTO for the example in the first section is $-170/4 = -42.5$. The confidence of DO is $-220/2 = -110$, under the convention that an alternative not explicitly listed by the classifier has a confidence of—200. The confidence of DIMENSION can be taken as minus infinity, simply because length considerations make it an impossible candidate. The actual implementation of the analyzer differs from the above description only in that confidences of the various control words are compared against a threshold sequentially, and a decision is made if the threshold is exceeded.

Statement analysis

After a P-list representing a statement to be resolved has been identified by type, it is analyzed in order to isolate its natural subparts. The details of these analyses depend upon the statement type, but the following two principles are common to all:

(1) Since the combinatorial explosion is the root of all evil, find delimiters that break the P-list into smaller segments that can be handled by other programs.
(2) Since no single character is reliable, spread the risk in finding a delimiter over a segment of the P-list as long as possible.

A single example will suffice to convey the flavor of the approach. Consider the IF statement. It is syntactically demanded that every IF statement have the form IF[expression] integer$_1$, integer$_2$, integer$_3$. It is easy to strip off the IF from the front of the statement, but we would also like to partition the remainder into a bracketed expression and a list of three integers. The inreliability of single characters make it unwise to seek merely the last right bracket (our character set uses brackets instead of parentheses), so we resort to a more elaborate technique. We start at the tail of the P-list representing the statement, and step along toward the front looking among the alternatives for, successively, a digit, a comma, and a digit. Having found this triple once, we continue to step toward the front looking among alternatives for a second digit-comma-digit triple. When we find it a second time we tentatively declare that the second comma has been passed (reading from right to left) and step along again, now looking for a digit-right bracket pair. When we find this pair, we tentatively declare that the delimiter "right-bracket" has been found and analyze the expression and integer-triple separately. If either of these analyses fails, we assume we have not yet found the delimiter and continue to step along toward the front. If we reach the front of the list without finding the delimiter, then the analysis fails, and the first choice decisions are accepted by default.

Similar analyses are used for other statement types in order to isolate such typical FORTRAN constructions as identifiers, lists of expressions, index controls and the like. These constructions are themselves the subject of further analysis. A subsequent section describes the analysis of what is perhaps the most interesting construction, the arithmetic expression.

The identifier table

Our semantic analysis is concerned only with the multiple appearance of the same identifier in a typical FORTRAN program. While rudimentary, this analysis has proven very successful and is used extensively. The analysis consists of three phases: constructing a table of identifiers (more precisely, the P-lists representing presumed identifiers), clustering the table, and finally using it to resolve FORTRAN statements. Each of these phases will be described in turn.

As currently implemented, assembly of the identifier table begins by restricting attention to statements rich in identifiers. For our purposes, we consider DIMENSION, COMMON, and the various input-output statements as our potential sources of identifier names. As a typical example, let us consider the extraction of identifiers from a COMMON statement. This statement consists of the word COMMON followed by a list of identifiers. One method of finding these identifiers is to search for possible commas in the P-list representing the statement and to assume that everything between commas is an identifier. This approach has proven unreliable because it places too much reliance on single characters. A more satisfactory algorithm searches the P-list exhaustively for all possible occurrences of a string of the form "alphanumeric-comma-alphabetic-alphanumeric . . . alphanumeric-comma-alphabetic." If such a string has a sufficiently high confidence, the segment of the P-list between the two commas is declared to represent an identifier and is added to the table. This algorithm is quite reliable because the confidences of at least five successive characters are computed even if the identifier has length one.

The second phase of semantic analysis involves "clustering" all identifiers of the same length. Clustering accomplishes two things: it prevents the same

identifier from appearing in several slightly different forms in the final result, and it allows us to correct identifier errors even in statements that contributed to the table. The clustering algorithm groups together all P-lists in the identifier table having at least one common alternative for each character. If, for each character, the common alternatives have a sufficiently high average confidence, then all the P-lists in the group are replaced by a single P-list having only the common alternatives. As an example, suppose the identifier table contained the following entries (we suppress the associated confidences for readability):

Entry[1]	*Entry*[2]
N A H E	W R M E
W M S	N A F
L	M
	X

The clustering algorithm would produce the single result

N A M E
W

if the average confidence of the two N's is higher than that of the two W's.

The final phase of identifier analysis is concerned with using the identifier table to resolve ambiguities in the text, and it is divided into two steps: finding a candidate segment of a P-list and matching the segment against the identifier table. Candidate segments of a P-list are found by essentially the same algorithm that assembled the table in the first place; an exhaustive search is made to find segments of the P-list that confidently might represent some identifier. When such a segment is found, it is compared against the table to find the best match. If the match is sufficiently good, the segment of the orginal P-list is replaced by a new segment having only a single alternative for each character.

The match of P-list against table of identifiers fulfills two needs. The obvious advantage is that it (presumably) results in a lower error rate in the final answer. Less obvious, perhaps, is the important reduction in the combinatorial complexity that is achieved by having only a single alternative for each element in a string of characters. We might mention here that one could certainly construct a table of labels as well as a table of identifiers, but this has not yet been implemented.

Resolution of arithmetic expressions

Each FORTRAN construction, such as identifiers, lists of integers, expressions, etc., is the subject of separate analysis. Of these, the analysis of expressions is probably the most interesting because of their complexity and variety. Further, many of the techniques used have been applied in the analysis of simpler constructions.

The expression analyzer has at its disposal a number of techniques which it applies in a fixed sequence. These techniques, in order of application, are (1) an identifier match with the table of identifiers, (2) a compression operation to reduce the combinatorial complexity, (3) a procedure to increase local consistency, (4) a partition of the P-list representing the expression into segments that might represent subexpressions, (5) an exhaustive resolution of the subexpressions, and (6) a reconstruction operation to "uncompress" the final answer. Suppose, then, that a P-list alleged to represent an expression has been obtained. Let us trace the action of the expression analyzer.

The first step is a matching operation against the table of identifiers. As previously described, this operation replaces appropriate segments of the P-list by new segments having only a single alternative for each original character. Once this is done the semantic ability of the analyzer is exhausted, so we take the second step of compressing the P-list. This is done to reduce combinatorial complexity by eliminating syntactically equivalent alternatives. Thus, if there are several letters as alternative to a single character, they may all be replaced by a single generic "X," with some associated confidence. Specification of the associated confidence is a little ticklish, but some heuristic arguments suggest that a good choice is the maximum confidence of the alternative letters. The same procedure is used for digits. Special characters, such as + and $, are left unchanged. As an example, the compression operation would convert the following P-list (suppressing confidences)

A	B	C	+	F	U	N
U	V	W	Q	1	2	3
R		S	1	4		5
						6
						Z

to a compressed version:

X	X	X	+	X	X	X
			X	1	1	1
			1			

The third operation of the expression analyzer is an investigation of local legality. It is interesting to note that, loosely speaking, an arbitrary string of alphanumeric characters can fail to be a legal expression because of either purely local or purely global illegalities.

Global illegalities cannot be detected by inspecting fixed-length segments of the string; typically they result from such things as bracket mismatches. Local illegalities can be detected by inspecting fixed-length segments, the simplest types arising from the juxtaposition of only two characters in an illegal manner, e.g., "* /" or "+]." The local legality check simply verifies that the highest confidence alternatives for pairs of consecutive characters are in fact legal pairs. If an illegal pair is found, then we consider not just that pair of characters, but the 4-tuple of characters centered on the illegal pair. Dynamic programming is then used to select alternatives that produce the legal 4-tuple of highest confidence. The confidences of these alternatives are increased enough to make them first choices. The choice of considering four consecutive characters is a compromise between the desire to make decisions on a global basis and the constraints of combinatorics. The basic advantages of the local check are its speed of operation and conservative nature. It often corrects some errors and never introduces fatal new ones.

The fourth step in the analysis of expressions is the breakdown of long P-lists into shorter ones. We select the operators +, −, /, and* as potential delimiters for partitioning the list. In other words, any character position having one of the above operators among its alternatives is a potential delimiter. A tenatative selection of delimiters is made on the basis of the relative confidences of the potential delimiters and their alternatives, and the segments of the P-list between these delimiters are examined. If each segment can be made into a legal subexpression (by means of dynamic programming) then the segments are strung together and the original expression is resolved. Otherwise, a different selection of tentative delimiters is made. There are a number of pitfalls in this operation. First, consider the expression A + FUN[X + Y]. This is certainly a legal expression, but the segment "FUN[X" delimited by the two plus signs is not a legal subexpression. This is annoying, but not fatal, since a subsequent iteration will presumably partition the expression as A and FUN[X + Y]. A second pitfall is exhibited by the following P-list (as usual, suppressing the associated confidences):

$$\begin{matrix} X\ X + + X \\ X \end{matrix}$$

If the first plus sign were chosen as the delimiter we would have the two simple legal subexpressions X X and + X as first choices of each segment of the P-list, but the concatenation of the two with another plus sign is illegal. This pitfall can be avoided by making a final legality check which, if not satisfied, forces a new selection of potential delimiters. Thus although the method of partitioning expressions is far from perfect, it works well in many cases and does serve to reduce the combinatorial explosion. The partitioning method described has proven useful in many other situations, e.g., partitioning a list of integers by means of commas, etc.

An alternative method of resolving arithmetic expressions is to do a left-to-right parse of the first choices of the P-list until an error is detected. The utility of this method depends largely on the proximity of error commision and error detection. If the two are nearly adjacent a parse could be very useful. Consider, however, a P-list in which the first choice for the first character is an erroneous left bracket. Detection of the error might well occur at the end of the expression, yielding very little useful information. In any event, a parse can tell only that an error has been committed; the localization and correction of the error must be accomplished by other means.

The last step in the resolution of expressions is to "uncompress" the answer. (Recall that all letters appear generically as "X" and all numbers as "1.") The reconstruction is accomplished in a straightforward manner by comparing the compressed answer with the original P-list and replacing each "X" by the highest confidence letter (and similarly for digits).

Summary of analyzer structure

We conclude this section with a summary description of the operation of the context-directed analyzer. The analyzer has a two-pass structure. The first pass determines the type of each statement (more precisely, of the statement represented by the P-list) and produces a clustered table of identifiers. The second pass accomplishes the resolution of each statement—the detection and correction of erroneous first choice alternatives. In this pass each statement is partitioned into subparts

CHARGE NO.

COLS. 1-10	COLS. 11-20	COLS. 21-30	COLS. 31-40	COLS. 41-50
COMMON AGE, WEIGHT, AGEMEAN, WTMEAN, COV				
DIMENSION AGE(1000), WEIGHT(1000)				
READ 100, IFLAG, NORE				
100 FORMAT(2I10)				
IF(NORE)16,5,5				
5 READ 101, AGE, WEIGHT				
101 FORMAT(F10.2)				
GO TO (16,24,30), IFLAG				
10 DO 11 I=1,1000				
11 WEIGHT(I)=AGE(I)				
GO TO 38				
24 DO 21 I=1,1000				
21 AGE(I)=WEIGHT(I)				
38 CALL AVERAGE(1000,AGEMEAN)				
CALL AVERAGE(1000,WTMEAN)				
COV=0.				
DO 50 I=1,1000				
50 COV=COV+(AGE(I)-AGEMEAN)*(WEIGHT(I)-WTMEAN)				
COV=COV/1000				
TYPE 102, IFLAG, COV				
102 FORMAT(I3,F10.5)				
GO TO				
60 STOP				
END				

FIGURE 1—A hand-printed FORTRAN program

```
      C0MM0N AGE,WESGHT,AGEMEAN,WTMEAN,C0V
      DIMENSI0N AGE[100],UEIGHT[100]
1     READ 100,IFLAG,M0KE
100   F0RM/T[2I8]
      IF[MER=]60,5,5
5     RKAD 101,AGE,WEIGHT
101   FORMATCF10.2]
      G0 T0 [1N,=0130],IFLAG
10    D7 11 I=1,100
1/    WEIGH=[I]=AGECS]
      G3 T0 10
20    D0 21 I=1,100
21    AGE[I]=WEIGHT[I]
70    CALL AVE[AGE,100,AGEMEAN]
      [ALL AVE[WEIGHT,100,WTMEAN]
      [0V=0.
      GD 50 I=1,100
50    C0V = C0V+[AGE[I]-AG[HCAN]*[WEIGHT[I]-UTMEAN]
      C0V=C0V/100.
      TYPE 102,IFLDG,C0V
102   F0RMAT[I8,F10.5]
      G0 T0 1
60    =T0F
      END
```

TABLE I—First choice of classifier

based on the syntactic requirements of the respective statement types. Each subpart consists of one of a small number of constructs, e.g., an expression or a list of integers. The analysis of each construct often begins with a match against the table of identifiers. If the resulting P-list is combinatorially simple it is analyzed exhaustively by dynamic programming. Otherwise, the P-list is further partitioned into segments by means of delimiters and the segments are examined. Before any exhaustive analysis is made the P-list is compressed to reduce the number of alternatives. The final answer is reconstructed from the compressed answer by comparison with the uncompressed P-list.

Example

Source data

This section describes an experiment illustrating the operation of the context-directed analyzer. The source data for this example came from the hand-printed FORTRAN program shown in Figure 1. The characters on the coding sheet shown were scanned, preprocessed, and classified by the methods described by Munson.[1] Because we purposely wanted data with a moderate error rate, we chose to use only topologically-derived features. Using these features, Munson had previously obtained a nine percent error rate on other data by the same writer. The results on this data were very similar, with 38 out of the 410 characters misclassified for an error rate of 9.3 percent. The particular errors made are shown underlined in Table 1. It is interesting to note that about one-third of these classification errors would not have been detected by purely syntactic methods.

These error rates correspond to the first choice responses of the character classifier, a linear machine. The linear machine classifies a character by computing dot products between a feature vector and 46 stored weight vectors, one for each of the 46 categories. The first choice response is the category corresponding to the largest dot product. The context-directed analyzer uses these dot products to determine alternative choices for each character, together with a measure of confidence for each alternative. The confidences C_i are obtained by normalizing the dot products according to

$$C_i = \frac{S_i - S_{max}}{S_{max}} \qquad i = 1, \cdots, 46,$$

where S_i is the i^{th} dot product, and S_{max} is the largest dot product observed for all of the characters in the source program.

Since the correct category for the character is usually included among the choices having high confidence, it is not necessary to consider every alternative for every character. An empirical study showed that almost invariably the correct category was among those alternatives whose dot products were at least half of the maximum dot product for that character. Thus, only those characters whose confidences met this condition were included in the list of alternatives. This typically reduced the number of alternatives from 46 to 4 or 5, with occasionally only 1, and never more than 10. The price for this simplification was an occasional failure to include the correct category, which was the case for the five doubly-underlined characters in Table 1. Although this introduces extra problems, the reduction in combinatorial complexity is worth the price.

Statement identification

As mentioned previously, the analyzer is organized as a two-pass program. During the first pass, the type of each statement is determined and variable names are collected for the construction of the identifier table. Statement identification was done by comparing the beginning of each statement with the "control" words, IF, DO, READ, etc. For example, the alternatives and the corresponding confidences for the first part of the sixth statement were as follows:

R	K	A	D	−25	−65	−28	−42
A	[	R	O	−49	−65	−62	−52
Z	E		V	−62	−68		−61
	U				−69		
	R				−77		
	L				−81		
	Z				−81		

The average confidence for the first choice selection RKAD was −40. The average confidence for READ was −41, which was sufficiently high to identify the state-

ment as a READ-statement. This matching procedure correctly identified all but one of the 24 statements, including two cases in which the correct category of a control word character was not included in the list of alternatives. However, absence of the correct category caused the ninth statement, a DO–statement, to be erroneously identified as the arithmetic-assignment statement D711I = 1,100. Subsequent analysis failed to resolve this as a legal arithmetic-assignment-statement, however, and the result of this failure condition was that first choice decisions from the classifier were accepted as the final output.

The identifier table

During the first pass, all COMMON, DIMENSION, and input/output statements were inspected to collect potential variable names. This operation was allowed to be somewhat liberal, since the inclusion of spurious identifiers is less harmful than the exclusion of actual identifiers. For example, in the last TYPE-statement the input/output list had the following alternatives:

I	F	L	D	G	,	C	O	V
[		[	A	6		[	D	W
S		6	M	+			Q	U
K		2	N	U			B	+
			,				G	O
			1				P	

Because the fifth-choice comma had a fairly high confidence, the program found IFL and G as well as IFLDG and COV as possible variable names. While there is a danger that these fragments might have accidentally matched similar names elsewhere in the program, no such matches occurred. One reason is that long names are tried before short names when the identifier table is used, and this prevents the premature discovery of erroneous matches with short fragments. Another is that completely accidental matches involving names of length greater than three or four are highly unlikely.

The search for possible variable names yielded the following (first choice) possibilities:

G	AGC	MOK[	IFLAG	WEIGHT	AGEMEAN
	COV		IFLDG	WTMCAN	
	AGE			UEIGHT	
	AGE			WEIGHT	
	IFL				
	COV				

Even in this simple example the need to cluster the identifier table is clear, since (a) four names were found more than once, and (b) three of these appeared with different first choice spellings. Clustering reduced the identifier table to the following first choice possibilities:

G	AGE	MOKE	IFLAG	WEIGHT	AGEMEAN
	COV			WTMCAN	
	IFL				

Of these names, two were spurious (G and IFL), but caused no trouble. Two were wrong (MOKE and WTMCAN), but since only one representative of each was found, they could not be fixed. The remainder (AGE, COV, IFLAG, WEIGHT, and AGEMEAN) were correctly clustered.

Statement analysis

During the second pass, each statement was resolved in turn. Since each different type of statement had to be treated differently, a complete description of how this was accomplished would be tedious. However, the spirit of our procedures can be conveyed by considering the resolution of the long arithmetic-assignment statement. For this statement, the first choices of the classifier were

```
50  COV = COV + [AGE[I] − AG[HCAN] *
          [WEIGHT[I] − UTMEAN] .
```

As with all statements, the label field (columns 1 to 5) was inspected first. Its resolution was trivial, since the first choices were legal. Attention then shifted to the statement field. Starting in column 7, a search was begun for a possible equals sign to be used to break the statement into a tentative variable and a tentative expression. (Had later procedures failed to resolve either of these parts, the search would have been resumed for a possible equals sign further to the right.)

The first character found having an equals sign for an alternative was, in fact, the correct equals sign. At this point, the first step was to resolve the left-hand side of the statement. Since the tentative variable, COV, could have been either a simple identifier (scalar variable) or an identifier followed by a bracketed list of expressions (array variable), a search was begun for a string of the form "alphanumeric, left-bracket." No such string was found, of course, and the tentative variable was declared to be just an identifier of length three. A search through the corresponding part of the identifier table produced a match, and COV was accepted for the name.

The next step was to resolve the expression. Here an exhaustive search of the expression for candidate identifiers was begun at once. Each candidate found was matched against appropriate length entries in the identifier table. This procedure produced five matches, and changed the first choices for the expression from

```
COV + [AGE[I] − AG[HCAN] * [WEIGHT[I]
          − UTMEAN]
```

to

$$\text{COV} + [\text{AGE[I]} - \text{AGEMEAN}] * [\text{WEIGHT[I]} - \text{WTMCAN}],$$

which, even though it contains the error in WTMEAN, is a syntactically valid expression. Thus, in this case the expression was resolved by the first operation, the use of the identifier table. The remaining operations, which have been very useful in other instances, were not needed, and hence were not performed. Since both the variable and the expression were now resolved, these parts were joined by an equals sign and appended to the results of the label-field analysis to yield the final resolution of the statement.

When similar procedures were applied to the other 23 statements, 28 of the 38 errors were corrected, reducing the error rate from 9.3 percent to 2.4 percent. The final output of the analyzer is shown in Table 2, where the 10 remaining errors are underlined. Three of these errors were due to the appearance of WTMCAN rather than WTMEAN in the identifier table, and three more were due to other problems with identifiers: MOKE, MERE, and S. A better method of using the identifier table, in which a final determination of variable names is postponed until all matches are made, would no doubt yield improved results.

Of the remaining four errors, one was in a FORMAT-statement, one in the DO-statement control word, and two involved labels. The FORMAT error was due to the fact that we have yet to implement that part of the program that resolves FORMAT statements. The DO error was caused by the missing alternative, and its correction would require the use of much more sophisticated methods for identifying statement types.Both label errors, however, could easily be cured by using a table of labels similar to the table of identifiers. Thus, roughly half of the 10 uncorrected errors could be resolved by relatively straightforward additions to our present program; the remainder would be difficult indeed to fix.

DISCUSSION

This paper has been concerned with techniques for using context to detect and correct character recognition errors. A few concluding remarks and observations about these techniques and their implementation are in order.

Our first observation is that the addition of new techniques to the context-analyzer program can continue virtually without limit. Many of these additions are straightforward. For example, tables of library subroutine names or statement labels could be incorporated and used in an obvious fashion. Other strategies, which humans employ with remarkably little effort, are much more difficult to implement. For example, the determination of statement type is currently made by matching the leading portion of the P-list against the various control words. A short control word results in a greater risk that the statement type will be misidentified, yet a human easily identifies statement types by their general appearance or gross structure, as well as by the (possibly misclassified) control word. This appreciation of global structure has been one of the more difficult abilities to give the analyzer.

Another observation is that the basic strategy employed by the analyzer should change with variations in the error rate of the input data. The support for this observation rests on intuitive, rather than experimental grounds, but it seems clear that elaborate procedures that may be required for very poor data are unnecessarily inefficient on very good data. While the present analyzer can cope with a certain amount of error-rate variability by automatic adjustment of thresholds, there is no provision to change the basic nature of the operations as a function of the quality of the input data.

A third observation is that there will always exist FORTRAN programs that are unlikely to be resolved successfully. One need only consider the contrary programmer who defines three separate variables as SS5S5, S5S5S, and S5SS5 to appreciate this. Whenever there can be errors in the input, there is a chance of errors in the output. In a practical system, one would want to provide the user with more than the final decision of the recognition system. For example, diagnostic messages could be given to aid the user in finding and correcting errors, whether they were committed by the classifier, the analyzer, or the user himself.

It is difficult to assess the usefulness of our techniques on the basis of an exploratory investigation. A

```
      COMMON AGE,WEIGHT,AGEMEAN,WTMCAN,COV
      DIMENSION AGE[100],WEIGHT[100]
1     READ 100,IFLAG,MOKE
100   FORMAT[2I8]
      IF[MERE]60,5,5
5     READ 101,AGE,WEIGHT
101   FORMAT[F10.2]
      GO TO [10,20,30],IFLAG
10    D7 11 I=1,100
11    WEIGHT[I]=AGE[S]
      GO TO 30
20    DO 21 I=1,100
21    AGE[I]=WEIGHT[I]
70    CALL AVE[AGE,100,AGEMEAN]
      CALL AVE[WEIGHT,100,WTMCAN]
      COV=0
      DO 50 I=1,100
50    COV=COV+[AGE[I]-AGEMEAN]*[WEIGHT[I]-WTMCAN]
      COV=COV/100.
      TYPE 102,IFLAG,COV
102   FORMAT[I8,F10.5]
      GO TO 1
60    STOP
      END
```

TABLE II—Final output of analyzer

thorough evaluation of the performance of the analyzer can be made only by testing it on a large number of FORTRAN programs produced by a variety of authors. Unfortunately, we were unable to undertake a data-processing project of this magnitude.

An equally difficult question concerns the extendability of the reported techniques to other problem domains. These techniques can be characterized by three qualities: risk-spreading in decision making, partitioning of a large decision problem into a hierarchy of subproblems, and continual checking of internal consistency. It seems clear that our basic approach applies more or less directly to other programming languages, and perhaps could be used with natural language in tightly constrained situations. The conjecture that the general approach, at least, can be applied in more general problems has a certain piquancy, but it remains only a conjecture.

We have, however, been able to achieve a substantial reduction in error rate for a particular application. In our opinion, it would have been difficult to obtain a comparable improvement by applying more conventional context analysis methods which do not take advantage of the special nature of the problem.

AKNOWLEDGMENTS

The authors wish to thank their colleagues at Stanford Reasearch Institute, and most particularly to thank Dr. John H. Munson for many stimulating and fruitful discussions.

This work has been supported by the United States Army Electronics Command, Fort Monmouth, New Jersey under Contract DA 28-043 AMC-01901(E).

APPENDIX

In this Appendix we derive the decision rule that classifies strings of alphanumeric characters in an optimal (minimum probability of error) fashion. By appropriately interpreting our result, we arrive at the decision rule described in the text.

Suppose that we are given some string of n characters to classify. Our problem is to determine a string of n categories that minimizes the probability of misclassification. If we let the vector X_i denote the set of measurements made on the i^{th} character and θ_i denote the category selected for the i^{th} character, then it is well known from Bayesian decision theory that the minimum probability of error is achieved by the following rule:

Select the categories $\theta_1, \cdots, \theta_n$ which maximize the posterior probability $p(\theta_1, \cdots, \theta_n|X_1, \cdots, X_n)$.

In other words, the posterior probability is computed for every possible assignment of θ_1 through θ_n, and the most probable assignment is taken as the decision.

By Bayes' law of inverse probabilities we can write the posterior probability as

$$p(\theta_1, \cdots, \theta_n|X_1, \cdots, X_n) = \frac{p(X_1, \cdots, X_n|\theta_1, \cdots, \theta_n)\, p(\theta_1, \cdots, \theta_n)}{p(X_1, \cdots, X_n)}. \quad (1)$$

This shows that the computation of the posterior probability depends upon both the prior probability $p(\theta_1, \ldots, \theta_n)$ and the conditional probability $p(X_1, \ldots, X_n|\theta_1, \ldots, \theta_n)$. To simplify the computation of the conditional density, we make the reasonable assumption that the manner in which a character is formed depends only upon the category of the character and not on the categories or the measurements of any surrounding character. This assumption is equivalent to an assumption of conditional independence, namely that

$$p(X_1, \cdots, X_n|\theta_1, \cdots, \theta_n) = \prod_{i=1}^{n} p(X_i|\theta_i). \quad (2)$$

At this point we must make some assumptions about the performance of the character classifier that provides the input to the context-directed analyzer. If this classifier were designed for the optimal classification of characters without regard to context it would compute, for every θ_i, the posterior probability

$$p(\theta_i|X_i) = \frac{p(X_i|\theta_i)\, p(\theta_i)}{p(X_i)}.$$

During the design of the classifier it was tacitly assumed that all classes are equally likely a priori, so that $p(\theta_i) = 1/46$. We therefore make the bold assumption that the classifier computes, for all 46 values of θ_i,

$$p^*(\theta_i|X_i) = \frac{p(X_i|\theta_i)\, 1/46}{p(X_i)}. \quad (3)$$

Substituting (3) and (2) into (1), we obtain

$$p(\theta_1, \cdots, \theta_n|X_1, \cdots, X_n) = \frac{p(\theta_1, \cdots, \theta_n)}{p(X_1, \cdots, X_n)} \prod_{i=1}^{n} 46\, p(X_i) p^*(\theta_i|X_i).$$

Now for given measurements $X_1, \ldots, X_n$ we are interested in maximizing this quantity over $\theta_1, \ldots, \theta_n$ so we can ignore constants and factors depending solely on X_i and obtain the following optimal compound decision rule:

Select the categories $\theta_1, \cdots, \theta_n$ for which

$$p(\theta_1, \cdots, \theta_n) \prod_{i=1}^{n} p^*(\theta_i|X_i) \text{ is maximum.}$$

We can, of course, take any monotonic function of this quantity and maximize it instead. Taking logarithms we can select the $\theta_1,\ldots,\theta_n$ which maximize

$$\log p(\theta_1, \cdots, \theta_n) + \sum_{i=1}^{n} \log p^* (\theta_i|X_i) .$$

If we define $\log p^* (\theta_i|X_i)$ as being the confidence that the measurements X_i indicate class θ_i, then we may reasonably define the confidence of the string to be

$$\sum_{i=1}^{n} \log p^*(\theta_i|X_i) .$$

The optimal decision rule, then, computes the confidence of each string of length n, biases each string confidence by adding the logarithm of the prior probability of the string, and selects as the answer that string having the highest biased confidence.

REFERENCES

1 J MUNSON
Experiments in the recognition of hand-printed text: Part I—Character recognition
In this volume

2 B GOLD
Machine recognition of hand-sent Morse code
IRE Trans on Information Theory Vol IT-5 pp 17–24 March 1959

3 W W BLEDSOE J BROWNING
Pattern recognition and reading by machine
Proc EJCC pp 225–232 Dec 1959 Also in *Pattern Recognition* L Uhr Ed pp 301–316 Wiley New York 1966

4 L D HARMON
Automatic reading of cursive script
In Optical Character Recognition Fischer et al Eds pp 151–152 Spartan Washington DC 1962

5 A W EDWARDS R L CHAMBERS
Can a priori probabilities help in character recognition
J ACM Vol 11 pp 465–470 October 1964

6 G CARLSON
Techniques for replacing characters that are garbled on input
AFIPS Conf Proc Vol 28 pp 189–192 Spring Joint Computer Conference 1966

7 C K McELWAIN M B EVENS
The degarbler—A program for correcting machine-read morse code
Information and Control Vol 5 pp 368–384 1962

8 C M VOSSLER N M BRANSTON
The use of context for correcting garbled English text
Proc ACM 19th National Conference paper D2 4–1 D2 4–13 1964

9 K ABEND
Compound decision procedures for pattern recognition
Proc NEC Vol 22 pp 777–780 1966

10 K ABEND
Compound decision procedures for unknown distributions and for dependent states of nature
In *Pattern Recognition* L Kanal Ed Thompson Book Co Washington DC 1968

11 J RAVIV
Decision making in Markov chains applied to the problem of pattern recognition
IEEE Trans on Info Thy Vol IT-13 pp 536–551 October 1967

12 R O DUDA P E HART J H MUNSON
Graphical-data-processing research study and experimental investigation
Fourth Quarterly Report Contract DA 28-043 AMC-01901 (E) SRI Project ESU 5864 Stanford Research Institute Menlo Park Calif March 1967