Error-Correcting Output Codes: A General Method for Improving Multiclass Inductive Learning Programs

Thomas G. Dietterich and **Ghulum Bakiri**
Department of Computer Science
Oregon State University
Corvallis, OR 97331-3202

Abstract

Multiclass learning problems involve finding a definition for an unknown function $f(\mathbf{x})$ whose range is a discrete set containing $k > 2$ values (i.e., k "classes"). The definition is acquired by studying large collections of training examples of the form $\langle \mathbf{x}_i, f(\mathbf{x}_i)\rangle$. Existing approaches to this problem include (a) direct application of multiclass algorithms such as the decision-tree algorithms ID3 and CART, (b) application of binary concept learning algorithms to learn individual binary functions for each of the k classes, and (c) application of binary concept learning algorithms with distributed output codes such as those employed by Sejnowski and Rosenberg in the NETtalk system. This paper compares these three approaches to a new technique in which BCH error-correcting codes are employed as a distributed output representation. We show that these output representations improve the performance of ID3 on the NETtalk task and of backpropagation on an isolated-letter speech-recognition task. These results demonstrate that error-correcting output codes provide a general-purpose method for improving the performance of inductive learning programs on multiclass problems.

Introduction

The task of learning from examples is to find an approximate definition for an unknown function $f(\mathbf{x})$ given training examples of the form $\langle \mathbf{x}_i, f(\mathbf{x}_i)\rangle$. For cases in which f takes only the values $\{0,1\}$—binary functions—there are many algorithms available. For example, the decision tree methods, such as ID3 (Quinlan, 1983, 1986b) and CART (Breiman, Friedman, Olshen & Stone, 1984) can construct trees whose leaves are labelled with binary values. Most artificial neural network algorithms, such as the perceptron algorithm (Rosenblatt, 1958) and the error backpropagation (BP) algorithm (Rumelhart, Hinton & Williams, 1986), are best suited to learning binary functions. Theoretical studies of learning have focused almost entirely on learning binary functions (Valiant, 1984; COLT 1988, 1989, 1990).

In many real-world learning tasks, however, the unknown function f takes on values from a discrete set of "classes": $\{c_1, \ldots, c_k\}$. For example, in medical diagnosis, the function might map a description of a patient to one of k possible diseases. In digit recognition, the function maps each hand-printed digit to one of $k = 10$ classes.

Decision-tree algorithms can be easily generalized to handle these "multi-class" learning tasks. Each leaf of the decision tree can be labelled with one of the k classes, and internal nodes can be selected to discriminate among these classes. We will call this the *direct multiclass* approach.

Connectionist algorithms are more difficult to apply to multiclass problems, however. The standard approach is to learn k individual binary functions $f_1, \ldots, f_k$, one for each class. To assign a new case, $\mathbf{x}$ to one of these classes, each of the f_i is evaluated on $\mathbf{x}$, and $\mathbf{x}$ is assigned the class j of the function f_j that returns the highest activation (Nilsson, 1965). We will call this the *one-per-class* approach, since one binary function is learned for each class.

Finally, a third approach is to employ a *distributed output code*. Each class is assigned a unique binary string of length n; we will refer to these as "codewords." Then n binary functions are learned, one for each bit position in these binary strings. These binary functions are usually chosen to be meaningful, and often independent, properties in the domain. For example, in the NETtalk system (Sejnowski & Rosenberg, 1987), a 26-bit distributed code was used to represent phonemes and stresses. The individual binary functions (bit positions) in this code corresponded to properties of phonemes and stresses, such as "voiced," "labial," and "stop." By representing enough distinctive properties of phonemes and stresses, each phoneme/stress combination can have a unique codeword.

For distributed output codes, training is accomplished as follows. For an example from class i, the desired outputs of the n binary functions are specified by the codeword for class i. With artificial neural networks, these n functions can be implemented by the n output units of a single network. With decision trees,

n separate decision trees are learned, one for each bit position in the output code.

New values of $\mathbf{x}$ are classified by evaluating each of the n binary functions to generate an n-bit string s. This string is then compared to each of the k codewords, and $\mathbf{x}$ is assigned to the class whose codeword is closest, according to some distance measure, to the generated string s.

This review of methods for handling multiclass problems raises several interesting questions. First, how do the methods compare in terms of their ability to classify unseen examples correctly? Second, are some of the methods more difficult to train than others (i.e., do they require more training examples to achieve the same level of performance)? Third, are there principled methods for designing good distributed output codes?

To answer these questions, this paper begins with a study in which the decision-tree algorithm ID3 is applied to the NETtalk task (Sejnowski & Rosenberg, 1987) using three different techniques: the direct multiclass approach, the one-per-class approach, and the distributed output code approach. The results show that the multiclass and distributed output code approaches generalize much better than the one-per-class approach.

It is helpful to visualize the output code of a learning system as a matrix whose rows are the classes and whose columns are the n binary functions corresponding to the bit positions in the codewords. In the one-per-class approach, there are k rows and k columns, and the matrix has 1's only on the diagonal. In the distributed output code approach, there are k rows and n columns, and the rows of the matrix give the codewords for the classes.

From this perspective, the two methods are closely related. A codeword is assigned to each class, and new examples are classified by decoding to the nearest of the codewords. This perspective suggests that a better distributed output code could be designed using error-correcting code methods. Good error-correcting codes choose the individual code words so that they are well-separated in Hamming distance. The potential benefit of such error correction is that the system could recover from errors made in learning the individual binary functions. If the minimum Hamming distance between any two codewords is d, then $\lfloor (d-1)/2 \rfloor$ errors can be corrected.

The "code" corresponding to the one-per-class approach, has a minimum Hamming distance of 2, so it cannot correct any errors. Similarly, many distributed output codes have small Hamming distances, because the columns correspond to meaningful orthogonal properties of the domain. In the Sejnowski-Rosenberg code, for example, the minimum Hamming distance is 1, because there are phonemes that differ only in whether they are voiced or unvoiced. These observations suggest that error-correcting output codes could be very beneficial.

On the other hand, unlike either the one-per-class or distributed-output-code approaches, the individual bit positions of error-correcting codes will not be meaningful in the domain. They will constitute arbitrary disjunctions of the original k classes. If these functions are difficult to learn, then they may negate the benefit of the error correction.

We investigate this approach by employing BCH error-correcting codes (Bose & Chaudhuri, 1960; Hocquenghem, 1959). The results show that while the individual binary functions are indeed more difficult to learn, the generalization performance of the system is improved. Furthermore, as the length of the code n is increased, additional performance improvements are obtained.

Following this, we replicate these results on the ISOLET isolated-letter speech recognition task (Cole, Muthusamy & Fanty, 1990) using a variation on the back propagation algorithm. Our error-correcting codes give the best performance attained so far by any method on this task. This shows that the method is domain-independent and algorithm-independent.

A Comparison of Three Multi-class Methods on the NETtalk Task

The NETtalk Task

In Sejnowski and Rosenberg's (1987) NETtalk system, the task is to map from English words (i.e., strings of letters) into strings of phonemes and stresses. For example,

```
f("lollypop") = ("lal-ipap", ">1<>0>2<").
```

Where `"lal-ipap"` is a string of phonemes, and `">1<>0>2<"` is a string of stress symbols. There are 54 phonemes and 6 stresses in the NETtalk formulation of this task. Note that the phonemes and stresses are aligned with the letters of the original word.

As defined, f is a very complex discrete mapping with a very large range. Sejnowski and Rosenberg reformulated f to be a mapping g from a seven-letter window to a phoneme/stress pair representing the pronunciation of the letter at the center of the window. For example, the word "lollypop" would be converted into 8 separate seven-letter windows:

```
g("___loll") = ("l", ">")
g("__lolly") = ("a", "1")
g("_lollyp") = ("l", "<")
g("lollypo") = ("-", ">")
g("ollypop") = ("i", "0")
g("llypop_") = ("p", ">")
g("lypop__") = ("a", "2")
g("ypop___") = ("p", "<")
```

The function g is applied to each of these 8 windows, and then the results are concatenated to obtain the phoneme and stress strings. This mapping function g now has a range of 324 possible phoneme/stress pairs. This is the task that we shall consider in this paper.

The Data Set

Sejnowski and Rosenberg provided us with a dictionary of 20,003 words and their corresponding phoneme and stress strings. From this dictionary we drew at random (and without replacement) a training set of 1000 words and a testing set of 1000 words. It turns out that of the 324 possible phoneme/stress pairs, only 126 appear in the training set, because many phoneme/stress combinations make no sense (e.g., consonants rarely receive stresses). Hence, in all of the experiments in this paper, the number of output classes is only 126.

Input and Output Representations

In all of the experiments in this paper, the input representation scheme introduced by Sejnowski and Rosenberg for the seven-letter windows is employed. In this scheme, the window is represented as the concatenation of seven 29-bit strings. Each 29-bit string represents a letter (one bit for each letter, period, comma, and blank), and hence, only one bit is set to 1 in each 29-bit string. This produces a string of 203 bits (i.e., 203 binary features) for each window. Experiments by Shavlik, Mooney, and Towell (1990) showed that this representation was better than treating each letter in the window as a single feature with 29 possible values. Of course, many other input representations could be used. Indeed, in most applications of machine learning, high performance is obtained by engineering the input representation to incorporate prior knowledge about the task. However, an important goal for machine learning research is to reduce the need to perform this kind of "representation engineering." In this paper, we show that general techniques for changing the *output* representation can also improve performance.

The representation of the output classes varies, of course, from one multiclass approach to another. For the direct multiclass method, the output class is represented by a single variable that can take on 126 possible values (one for each phoneme/stress pair that appears in the training data). For the one-per-class approach, the output class is represented by 126 binary variables, one for each class.

For the distributed output code approach, we employ the code developed by Sejnowski and Rosenberg. We used the Hamming distance between two bit-strings to measure distance. Ties were broken in favor of the phoneme/stress pair that appeared more frequently in the training data. In Dietterich, Hild, and Bakiri (1990a), we called this "observed decoding."

The ID3 Learning Algorithm

ID3 is a simple decision-tree learning algorithm developed by Ross Quinlan (1983, 1986b). In our implementation, we did not employ windowing, CHI-square forward pruning (Quinlan, 1986a), or any kind of reverse pruning (Quinlan, 1987). Experiments reported in Dietterich, Hild, and Bakiri (1990b) have shown that these pruning methods do not improve performance. We did apply one simple kind of forward pruning to handle inconsistencies in the training data: If at some point in the tree-growing process all training examples agreed on the values of all features—and yet disagreed on the class—then growth of the tree was terminated in a leaf and the class having the most training examples was chosen as the label for that leaf (ties were broken arbitrarily for multiclass ID3; ties were broken in favor of class 0 for binary ID3).

In the direct multiclass approach, ID3 is applied once to produce a decision tree whose leaves are labelled with one of the 126 phoneme/stress classes. In the one-per-class approach, ID3 is applied 126 times to learn a separate decision tree for each class. When learning class i, all training examples in other classes are considered to be "negative examples" for this class. When the 126 trees are applied to classify examples from the test set, ties are broken in favor of the more-frequently-occurring phoneme/stress pair (as observed in the training set). In particular, if none of the trees classifies a test case as positive, then the most frequently occurring phoneme/stress pair is guessed. In the distributed output code approach, ID3 is applied 26 times, once for each bit-position in the output code.

Results

Table 1 shows the percent correct (over the 1000-word test set) for words, letters, phonemes, and stresses. A word is classified correctly if each letter in the word is correct. A letter is correct if the phoneme and stress assigned to that letter are both correct. For the one-per-class and distributed output code methods, the phoneme is correct if all bits coding for the phoneme are correct (after mapping to the nearest legal codeword and breaking ties by frequency). Similarly, the stress is correct if all bits coding for the stress are correct.

There are several things to note. First, the direct multiclass and distributed output codes performed equally well. Indeed, the statistical test for the difference of two proportions cannot distinguish them. Second, the one-per-class method performed markedly worse, and all differences in the table between this method and the others are significant at or below the .01 level.

Error Correcting Codes

The satisfactory performance of distributed output codes prompted us to explore the utility of good error-correcting codes. We applied BCH methods (Lin & Costello, 1983) to design error-correcting codes of varying lengths. These methods guarantee that the rows of the code (i.e., the codewords) will be separated from each other by some minimum Hamming distance d.

Table 2 shows the results of training ID3 with distributed error-correcting output codes of varying lengths. Phonemes and stresses were encoded separately, although this turns out to be unimportant.

Table 1: Comparison of Three Multi-class Methods

	Level of Aggregation (% Correct)				Tree Statistics		
						Average	
Method	Word	Letter	Phoneme	Stress	N	Leaves	Depth
Direct Multiclass	13.5	70.8	81.1	78.3	1	2652.0	73.0
One-per-class	8.7	66.7	76.4	74.5	126	34.9	10.5
Distributed	12.5	69.6	81.3	79.2	26	270.0	29.3

Table 2: Performance of Error-Correcting Output Codes

BCH Code				Level of Aggregation				Tree Statistics		
Phoneme		Stress		% Correct (1000-word test set)					Average	
n	d	n	d	Word	Letter	Phon.	Stress	N	Leaves	Depth
10	3	9	3	13.3	69.8	80.3	80.6	19	677.4	51.9
14	5	11	5	14.4	70.9	82.3	80.3	25	684.7	53.1
21	7	13	7	17.2	72.2	83.9	80.4	34	681.4	53.9
26	11	13	11	17.5	72.3	84.2	80.4	39	700.5	56.4
31	15	30	15	19.9	73.8	84.8	81.5	61	667.8	52.7
62	31	30	15	20.6	74.1	85.4	81.6	77	669.9	53.3
127	63	30	15	20.8	74.4	85.7	81.6	157	661.6	54.8

Columns headed n show the length of the code, and columns headed d show the Hamming distance between any two code words.

The first thing to note is that the performance of even the simplest (19-bit) BCH code is superior to the 26-bit Sejnowski-Rosenberg code at the letter and word levels. Better still, performance improves monotonically as the length (and error-correcting power) of the code increases. The long codes perform much better than either the direct multiclass or Sejnowski-Rosenberg approaches at all levels of aggregation (e.g., 74.4% correct at the letter level versus 70.8% for direct multiclass).

Not surprisingly, the individual bits of these error-correcting codes are much more difficult to learn than the bits in the one-per-class approach or the Sejnowski-Rosenberg distributed code. Specifically, the average number of leaves in each tree in the error-correcting codes is roughly 665, whereas the one-per-class trees had only 35 leaves and the Sejnowski-Rosenberg trees had 270 leaves. Clearly distributed output codes do not produce results that are easy to understand!

The fact that performance continues to improve as the code gets longer suggests that we could obtain arbitrarily good performance if we used arbitrarily long codes. Indeed, this follows from information theory under the assumption that the errors in the various bit positions are independent. However, because each of the bits is learned using the same body of training examples, it is clear that the errors are not independent. We have measured the correlation coefficients between the errors committed in each pair of bit positions for our BCH codes. All coefficients are positive, and many of them are larger than 0.30. Hence, there must come a point of diminishing returns where further increases in code length will not improve performance. An open problem is to predict where this breakeven point will occur.

Error-correcting Codes and Small Training Sets

Given that the individual binary functions require much larger decision trees for the error-correcting codes than for the other methods, it is important to ask whether error-correcting codes can work well with smaller sample sizes. It is well-established that small training samples cannot support very complex hypotheses.

To address this question, Figure 1 shows learning curves for the distributed output code and for the 93-bit error-correcting code (63 phoneme bits, 30 stress bits). At all sample sizes, the performance of the error-correcting configuration is better than the Sejnowski-Rosenberg distributed code. Hence, even for small samples, error-correcting codes can be recommended.

Replication in Isolated Letter Speech Recognition

To test whether error-correcting output codes provide a general method for boosting the performance of inductive learning algorithms, we applied them in a second domain and with a different learning algorithm. Specifically, we studied the domain of isolated letter speech recognition and the back propagation learning algorithm.

In the isolated-letter speech-recognition task, the "name" of a single letter is spoken by an unknown spea-

Table 3: Parameter Values Selected via Cross Validation

Configuration	# Hidden Units	Best TSS
one-per-class	78	10.50
30-bit ECC	156	142.66
62-bit ECC	156	161.76

ker and the task is to assign this to one of 26 classes corresponding to the letters of the alphabet. Ron Cole has made available to us his ISOLET database of 7,797 training examples of spoken letters (Cole, Muthusamy & Fanty, 1990). The database was recorded from 150 speakers balanced for sex and representing many different accents and dialects. Each speaker spoke each of the 26 letters twice (except for a few cases). The database is subdivided into 5 parts (named ISOLET1, ISOLET2, etc.) of 30 speakers each.

Cole's group has developed a set of 617 features describing each example. Each feature has been scaled to fall in the range $[-1, +1]$. We employed the `opt` (Barnard & Cole, 1989) implementation of backpropagation with conjugate gradient optimization in all of our experiments.

In our experiments, we compared the one-per-class approach to a 30-bit ($d = 15$) BCH code and a 62-bit ($d = 31$) BCH code. In each case, we used a standard 3-layer network (one input layer, one hidden layer, and one output layer). In the one-per-class method, test examples are assigned to the class whose output unit gives the highest activation. In the error-correcting code case, test examples are assigned to the class whose output codeword is the closest to the activation vector produced by the network as measured by the following distance metric: $\sum_i |act_i - code_i|$.

One advantage of conjugate-gradient optimization is that, unlike backpropagation with momentum, it does not require the user to specify a learning rate or a momentum parameter. There are, however, three parameters that must be specified by the user: (a) the starting random-number seed (used to initialize the artificial neural network), (b) the number of hidden units, and (c) the total-summed squared error at which training should be halted (this avoids over-training).

To determine good values for these parameters, we followed the "cross-validation" training methodology advocated by Lang, Waibel, and Hinton (1990). The training data were broken into three sets:

- The **training set** consisting of the 3,120 letters spoken by 60 speakers. (These are the examples in Cole's files ISOLET1 and ISOLET2.)
- The **cross-validation set** consisting of 3,118 letters spoken by 60 additional speakers. (These are the examples in files ISOLET3 and ISOLET4.)
- The **test set** consisting of 1,559 letters spoken by 30 additional speakers. (These are the examples in file ISOLET5.)

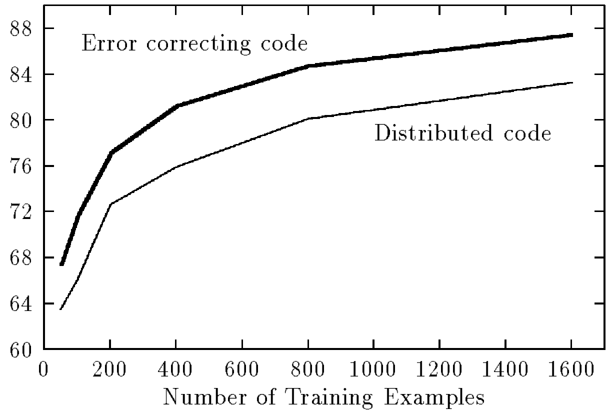

Figure 1: Learning curves showing % phonemes correct for the distributed output code and for the 93-bit error-correcting code (63 phoneme bits with $d = 31$, 30 stress bits with $d = 15$).

Table 4: Performance in the Isolated Letter Domain

Configuration	Actual TSS	% Correct	% Error
one-per-class	10.51	95.83	4.17
30-bit ECC	142.26	96.73	3.27
62-bit ECC	161.85	95.96	4.04

The idea is to vary the parameters while training on the training set and testing on the cross-validation set. The parameter values giving the best performance on the cross-validation set are then used to train a network using the *union* of the training and cross-validation sets, and this network is then tested against the test set. We varied the number of hidden units between 35 and 182, and, for each number of hidden units, we tried four different random seeds. Table 3 shows the parameter values that were found by cross validation to give the best results.

Table 4 shows the results of training each configuration on the combined training and cross-validation sets and testing on the test set. Both error-correcting configurations perform better than the one-per-class configuration. The results are not statistically significant (according to the test for the difference of two proportions), but this could be fixed by using a larger test set. The results are very definitely significant from a practical standpoint: The error rate has been reduced by more than 20%. This is the best known error rate for this task.

Conclusions

The experiments in this paper demonstrate that error-correcting output codes provide an excellent method for applying binary learning algorithms to multiclass learning problems. In particular, error-correcting output codes outperform the direct multiclass method, the one-per-class method, and a domain-specific distributed output code (the Sejnowski-Rosenberg code for the nettalk domain). Furthermore, the error-correcting output codes improve performance in two very different domains and with two quite different learning algorithms.

We have investigated many other issues concerning error-correcting output codes, but, due to lack of space, these could not be included in this paper. Briefly, we have demonstrated that codes generated at random can act as excellent error-correcting codes. Experiments have also been conducted that show that training multiple neural networks and combining their outputs by "voting" does not yield as much improvement as error-correcting codes.

Acknowledgements

The authors thank Terry Sejnowski for making available the NETtalk dictionary and Ron Cole and Mark Fanty for making available the ISOLET database. The authors also thank NSF for its support under grants IRI-86-57316 and CCR-87-16748. Ghulum Bakiri was supported by Bahrain University.

References

Barnard, E. & Cole, R. A. (1989). A neural-net training program based on conjugate-gradient optimization. Rep. No. CSE 89-014. Beaverton, OR: Oregon Graduate Institute.

Bose, R. C., & Ray-Chaudhuri, D. K. (1960). On a class of error-correcting binary group codes. *Inf. Cntl.*, 3, pp. 68-79.

Breiman, L, Friedman, J. H., Olshen, R. A., & Stone, C. J. (1984). *Classification and Regression Trees.* Monterey, CA: Wadsworth and Brooks.

Cole, R. Muthusamy, Y. & Fanty, M. (1990). The ISOLET spoken letter database. Rep. No. CSE 90-004. Beaverton, OR: Oregon Graduate Institute.

COLT (1988). Haussler, D. & Pitt, L. (Eds.) *COLT '88*, Cambridge, MA: Morgan Kaufmann.

COLT (1989). Rivest, R. L., Haussler, D., and Warmuth, M. K. (Eds.) *COLT '89: Proceedings of the Second Annual Workshop on Computational Learning Theory*, Santa Cruz, CA: Morgan Kaufmann.

COLT (1990). Fulk, M. A., and Case, J. (Eds.) *COLT '90: Proceedings of the Third Annual Workshop on Computational Learning Theory.* Rochester, NY: Morgan Kaufmann.

Dietterich, T. G., Hild, H., Bakiri, G. (1990a) A comparative study of ID3 and backpropagation for English text-to-speech mapping. *7th Int. Conf. on Mach. Learn.* (pp. 24–31). Austin, TX: Morgan Kaufmann.

Dietterich, T. G., Hild, H., Bakiri, G. (1990b) A comparison of ID3 and backpropagation for English text-to-speech mapping. Rep. No. 90-30-4. Corvallis, OR: Oregon State University.

Hocquenghem, A. (1959). Codes corecteurs d'erreurs. *Chiffres*, 2, pp. 147-156.

Lang, K. J, Waibel, A. H, & Hinton, G. E. (1990). A time-delay neural network architecture for isolated word recognition. *Neural Networks, 3*, 33-43.

Lin, S., & Costello, D. J. Jr. (1983). *Error Control Coding: Fundamentals and Applications.* Englewood Cliffs: Prentice-Hall.

Nilsson, N. J., (1965). *Learning Machines*, New York: McGraw Hill.

Quinlan, J. R. (1983). Learning efficient classification procedures and their application to chess endgames, in Michalski, R. S., Carbonell, J. & Mitchell, T. M., (eds.), *Machine learning, Vol. I*, Palo Alto: Tioga Press. 463–482.

Quinlan, J. R. (1986a). The effect of noise on concept learning. In Michalski, R. S., Carbonell, J. & Mitchell, T. M., (eds.), *Machine learning, Vol. II*, Palo Alto: Tioga Press. 149–166.

Quinlan, J. R. (1986b). Induction of Decision Trees, *Machine Learning,* 1(1), 81–106.

Quinlan, J. R., (1987). Simplifying decision trees. *Int. J. Man-Mach. Stud., 27*, 221–234.

Rosenblatt, F. (1958). The perceptron. *Psych. Rev., 65* (6), 386–408.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning internal representations by error propagation. In Rumelhart, D. E. & McClelland, J. L., (eds.) *Parallel Distributed Processing*, Vol 1. 318–362.

Sejnowski, T. J., and Rosenberg, C. R. (1987). Parallel networks that learn to pronouce English text. *Complex Syst.*, 1, 145–168.

Shavlik, J. W., Mooney, R. J., and Towell, G. G. (1990). Symbolic and neural learning algorithms: An experimental comparison. *Mach. Learn., 6*, 111–144.

Valiant, L. G. (1984). A theory of the learnable. *CACM, 27,* 1134–1142.