A version of this paper will appear in *Massively Parallel Artificial Intelligence,* H. Kitano (ed). Menlo Park, CA: AAAI Press/The MIT Press (1993).

Selecting Salient Features for Machine Learning from Large Candidate Pools through Parallel Decision-Tree Construction

Kevin J. Cherkauer
cherkaue@cs.wisc.edu

Jude W. Shavlik
shavlik@cs.wisc.edu

1-608-262-6613 voice, 1-608-262-9777 fax

Computer Sciences Department, University of Wisconsin-Madison
1210 W. Dayton St., Madison, WI 53706, USA

Keywords: artificial neural networks, computational biology, decision trees, inductive learning, parallel algorithms, protein folding

1 Introduction

The primary goal of machine learning research is to develop algorithms that enable computers to learn to perform various tasks. These can range anywhere from guiding a robot's motion through complex environments to predicting tomorrow's weather. The common goal which all machine learning systems share is that of substituting learning for explicit programming, in the hope that this may lead to easier, faster, and even better system construction than is possible through hand coding.

A major category of machine learning systems is that of *inductive classification systems.* Such systems learn to categorize observational instances, or *examples,* in some sensible way, either according to a predefined set of classifications (*supervised learning*) or a set of freely discovered regularities in the examples (*unsupervised learning*). This chapter is mainly concerned with inductive classification systems within the paradigm of supervised learning from examples.

Supervised learning algorithms examine sets of preclassified *training examples* and attempt to discover regularities that determine their classifications. After learning, they can then classify new examples from the same problem domain. We can evaluate system performance by measuring post-learning correctness on a set of *testing examples,* which are just previously unseen examples whose proper classifications are known.

In order for a system to learn from examples, we must represent the examples in some way it can understand. The abstract representation technique most often used is that of *feature vectors.* That is, each example is simply a fixed set, or vector, of features along with their instantiated values. We are concerned in this chapter with the choice of these features, or the *representation selection* problem.

For the purposes of describing the representation selection problem, we divide all features into two categories, *measured features* and *constructed features.* Measured features are those one obtains directly from the world through some sort of measurement process. These are generally chosen with an eye to the problem to be solved, although in some cases they may be taken from data gathered for other purposes. A primary desire is that the available set of measured features captures all the information about the world which is necessary to solve the target classification problem. For instance, if one wishes to train a learning system to predict the risk of individuals suffering heart disease in the next five years, one might decide to measure such features as age, sex, weight, blood pressure, serum cholesterol level, and so on. These features are chosen because one believes them to be those which are relevant to the problem; thus, they attempt to capture the information which will make accurate prediction possible.

There is no guarantee, however, that this will turn out to be the case. It may be that some of the features measured are irrelevant and so serve only to confound the learning. Worse, the measured features may not, in fact, include everything needed to solve the problem. For example, the amount of exercise a person gets may be an important factor in determining the incidence of heart disease, yet this information may not have been collected in the medical histories of the patients studied. Unfortunately, there is usually not much one can do about this in real-world problems short of going back and making more measurements. This process is in general costly and (as in the case of medical records spanning a long period of time) may even be impossible, so most of the time one will be "stuck," so to speak, with the features already collected. Because of the frequent infeasibility of gathering more measured features, we will here operate under assumption that we cannot do so.

Given a set of measured features, then, our problem is to make the best use of the information that they do in fact contain. At first glance, this seems like an easy problem to solve: simply encode these features in some straightforward way that is amenable to one's favorite learning algorithm and let it run. This answer is somewhat naive, however, because the *form* in which the information is presented can have a marked effect on the ease of solving the problem. To make an analogy, if a skilled author is attempting to explain a complicated point, he or she will probably do so in such a way as to minimize the reader's burden in understanding it. The presentation will be clear and concise, and can even turn a difficult concept into something which is grasped in a few moments. Alas, we all too often experience the other side of this coin: the writer may understand the concept perfectly but nonetheless produce a description which is opaque, convoluted, and inscrutable. Both articles may contain the same information, but this does not mean both are of equal value in transmitting it in a meaningful or useful way.

The same is true of representations for machine learning, where the representation is analogous to the written description and the learning algorithm is analogous to the reader. That an example representation contains useful information is no guarantee that it makes the problem easy to solve. On the contrary, it is just as possible for an example representation to be opaque and inscrutable as it is for an author to produce indigestible descriptions. The end result is the same: the clarity of the representation deeply affects the ease with which a concept can be learned. Thus, we wish to make our example representation as clear and "natural" for the given problem as possible. Since we have already ruled out the possibility of collecting more measured features in this attempt, we must move instead to the second

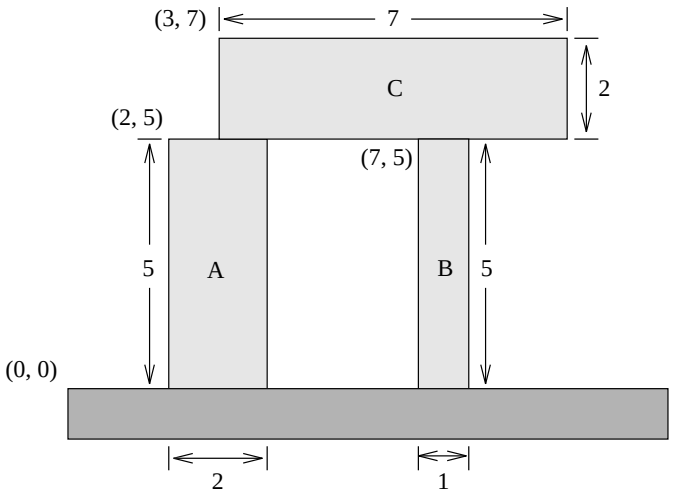

Figure 1: An arch.

type of feature mentioned above—the constructed feature.

Constructed features are what the name implies: new features which are constructed from the measured features. Research in so-called "constructive induction" (Michalski, 1983; Ragavan & Rendell, 1993; Matheus & Rendell, 1989; Pagallo & Haussler, 1990) deals with the creation of this type of feature, which can involve logical combinations, numerical comparisons, or even complex computations which yield a final single value. Ideally, we would like to construct new features which make the problem easier to solve. As an illustration, consider the problem of determining whether an arch is present in a drawing containing three rectangular blocks and a table. For simplicity, let us define an "arch" as any configuration in which two of the blocks are positioned vertically, with space between them, and support the third block in a horizontal position (e.g. as in Figure 1). Each particular block configuration constitutes one example which must somehow be represented to the system. In this limited scenario, we can completely represent the problem state by specifying, say, the (x, y) coordinates of the upper-left corner of each block and each block's *width* (horizontal extension) and *height* (vertical extension).

This seems like a plausible enough representation in that it is small and also contains, in principle, all the information needed to solve the problem. However, the form in which this information is expressed has substantial ramifications for learning. As mentioned earlier, the performance of inductive learning algorithms is intimately tied to the representation used for example description. Final classification accuracy may vary widely depending on this representation, even though other factors are held constant (e.g. Uberbacher & Mural, 1991; Blum & Rivest, 1992; Farber *et al.,* 1992; Craven & Shavlik, 1993a, 1993b), simply because the form of the representation introduces a bias as to what concepts are most easily expressed.

To continue with our simple arch example, note that it is not the absolute positions and dimensions of the blocks that are important in determining the quality called "arch," but rather the relationships which hold between them. For instance, it is essentially irrelevant that the y coordinate of block A is five. It is much more important that it is equal to block C's y coordinate minus C's height, because this indicates that A may plausibly support C

if their horizontal extensions overlap. Likewise, we are more interested in the fact that A's height is greater than its width (to establish a vertical orientation) than in the absolute values of either of these attributes.

For many machine learning algorithms (and for people too!), the particular sets of features used to describe examples can make the target concept either naturally intuitive or hopelessly obscure. Most people would certainly find the arch classification task easier if the features given are of the sort just discussed, e.g. "block A is vertical" and "block A supports block C," than if coordinates and sizes are specified instead. Unfortunately, the task of finding the "right" kinds of features for a given problem is very difficult, because even simple methods of feature construction can lead to combinatorially many new features. For instance, the set of all possible conjunctions of a given set of Boolean-valued features has an exponential number of members, since every subset of the original features could form a conjunct. If more complex constructions are allowed or if the initial data contains, say, numeric features, it is easy to see that an infinite number of new features can be constructed.

In the arch example, we can easily judge which constructed features are of the most utility, i.e. that a comparison between the width and height of a single block is relevant to the task whereas a comparison of the width of one block to the height of another is not. However, in more interesting problems such a priori intuition will probably be much more vague, so the relatively few important constructed features will be buried amidst the vast numbers of possible ones. Thus, the problem of finding a small set of highly informative constructed features is a daunting one—so much so that in fact most of the time little effort is put into the choice of a representation other than to encode the available measured features in some straightforward fashion and leave it at that.

The difficulty of the representation-choice problem, coupled with the potential rewards for its successful solution, provides the main motivation for our attempt to develop an automated parallel method of feature selection. Our system, DT-SELECT ("Decision Tree feature Selection"), searches in the space of measured and constructed features for representations that make particular learning problems easier. (It is so-named because its core is essentially an efficient parallel implementation of Quinlan's ID3 (1986) decision-tree-building algorithm.) Because of the combinatorial number of possible constructed features, we cannot hope to blindly stumble across treasures; we need to employ some kind of high-level guidance if we are to prevail. Nevertheless, just because there are so many possible features, once we have chosen an area of the space to examine we would like to explore it as thoroughly as possible in order to find the riches which may be hiding like a needle in a haystack.

DT-SELECT attempts to balance these two opposing needs. It allows the user to guide the search in directions which seem promising by specifying the *types* of features that should be investigated. Just as we had the intuition in the arch example that making certain sorts of comparisons would lead us to the discovery of predictive features for that problem, so too may we have similar intuitions in more complex domains. The problem here (unlike in the arch example) is that we probably will not know which particular instantiations of a certain type of feature are likely to contain the most information. It is here that the parallel search of DT-SELECT plays its part: given a set of user-specified feature types, DT-SELECT performs an exhaustive search of their instantiations to find economical sets of individually salient features which together capture the information needed to solve the problem. Thus human and machine offset each other's weaknesses, the human providing the

necessary intuitive guidance while the machine provides the thoroughness and accuracy to put it to use.

In the following section we lay out our desiderata for an ideal example representation, which our DT-SELECT algorithm then attempts to address. The remainder of the chapter describes DT-SELECT in detail and presents some initial experiments with its use in the molecular biology task of predicting the folded shapes of globular proteins.

2 In Search of the Ideal Representation

To make the task of inductive learning as easy as possible, we would ideally like to find a representation that simultaneously maximizes three desirable characteristics:

- Coverage
- Economy
- Transparency

A representation's *coverage* indicates the degree to which it captures, in principle, all the information needed to solve the given problem. For instance, the naive representation in our earlier arch example, in which the drawing was described in terms of the widths, heights, and (x, y) coordinates of the three blocks, exhibits complete coverage in that everything necessary to separate arches from nonarches is encoded in these values.

This representation is also laudable in its *economy,* the second of our three dimensions. Economy refers to the representation's compactness, or succinctness. We wish to avoid hampering the learning task with the inclusion of many features which represent nearly the same thing (i.e. *redundant* features). Indeed, it is economy in which the simple arch representation best excels, as it captures the entire problem state in a mere twelve numbers, none of which are derivable from others and all of which contain information needed to determine the correct classification.

However, this simple representation falls short on the third point, for it does not manifest *transparency.* By this we mean that the features in the representation do not present the information needed to separate the examples into their respective classes in a straightforward fashion which would make solving the problem easy. Instead, a learning algorithm using this representation is forced to somehow discover the salient higher-order relationships between the raw features—a feat that is made difficult by the combinatorial number of possibilities. A more transparent representation would provide a more direct route from features to classifications. The most extreme (and also unrealistic) form of transparency would specify the example classifications explicitly as an input feature, making problem solving trivial. (Transparency can be thought of as the inverse of Ragavan and Rendell's (1993) "blurring.")

In attempting to generate a representation which satisfies the three criteria of coverage, economy, and transparency, there are obviously tradeoffs that must be considered (e.g. coverage versus economy and economy versus transparency). As we have seen in the example of the arch, a representation that is strong in one or two of these aspects but poor in the others will not necessarily facilitate easy learning. DT-SELECT uses *decision trees* as a

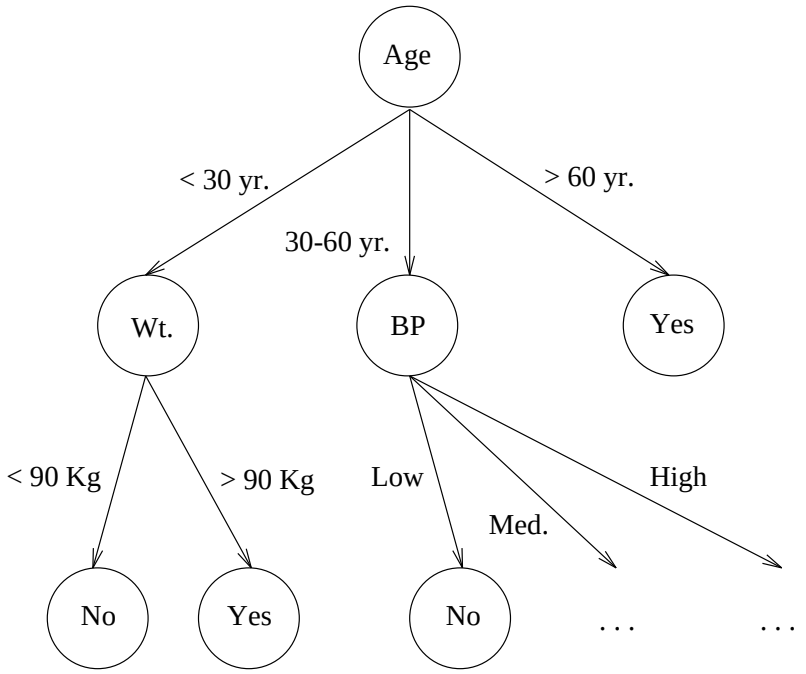

Figure 2: A fictional decision tree for predicting heart disease.

means of choosing sets of features having a balance of all three characteristics. The next section provides a quick tutorial describing what decision trees are and how they can be used to select features satisfying our three desiderata. Following that, we describe the complete DT-SELECT algorithm in detail.

3 Decision Trees

A decision tree is simply a tree data structure with the following properties:

- Each internal node is labeled with the name of a feature
- Each leaf is labeled with the name of an example category
- Every internal node has a set of at least two children, where the branches to the children are labeled with disjoint values or sets of values of that node's feature such that the union of these constitutes the set of all possible values for that feature

Thus, the labels on the arcs leaving a parent node form a partition of the set of legal values for the parent's feature. Figure 2 provides a simple illustration of part of a decision tree which might apply to our earlier heart disease example.[1] Leaves labeled "Yes" indicate people at high risk for heart disease, while those labeled "No" indicate low risk. Two subtrees at the lower right are not shown.

[1]The "information" about heart disease this example contains is fictional—so please do not become concerned!

Decision trees are commonly built inductively and used as classifiers (Quinlan, 1986). An example can be classified by a decision tree by starting at the root of the tree and, for each internal node visited, traversing the child branch whose associated set of values contains the value of that node's feature seen in the example. When a leaf is reached, its label gives the (hopefully correct) classification of that example. While DT-SELECT does construct decision trees, we are here interested in using them not as classifiers in themselves (though this is also a possibility) but instead as a means to a further end—that of feature selection.

The typical approach to building decision trees inductively is to pick as the root of the tree the feature whose values best separate a collection of training examples into their categories. These values define a partition of the training examples, where each subset contains examples having the same value for the chosen feature. Children of the root are then added to the tree by performing a similar feature-selection step recursively for each subset in the partition. The process stops when every training example set contains examples of only one class.

DT-SELECT uses this general approach to build a decision tree for a given set of training examples, except that the features chosen may be constructed as well as measured features. The features used by the final decision tree then constitute the new example representation which the system has selected.[2] Selecting sets of features in this fashion allows us to approximate all three representation ideals (coverage, economy, and transparency) in an efficient way. First of all, features are chosen specifically according to their transparency: the feature that is most predictive in isolation is always selected at each step.

Secondly, because each feature added to the tree partitions its training data better than any other, coverage (or ability to separate the data entirely) is increased steadily as the tree is built.

Finally, once a particular feature is chosen as the root of a subtree, other highly correlated (i.e. redundant) features will not be included in the body of the subtree since they have little power to further separate the subsets of training examples that the root feature has already defined. This provides a kind of "piecewise orthogonality" to the final set of features, lending it descriptive economy. In contrast, if we simply chose the k individually most informative features as our new representation, we might instead obtain a highly redundant feature set with poor discriminatory power—a problem which is especially acute if we wish, as we do, to select from a very large space of features. This is because many slightly different versions of the few best features would be chosen, rather than a mix of features which apply to different types of examples. Using decision trees for selection helps control this problem of highly correlated features.

Almuallim and Dietterich (1991) offer a different feature selection method, but its emphasis is on finding a *minimal* set of features which can separate the data. Their approach thus concentrates on optimizing economy while guaranteeing coverage, but it does nothing to insure transparency. Yet it is this last attribute which would seem to be the crux of representation selection: the primary reason we wish to construct new representations is to provide a transparency which will make solving the target problem easier. We postulate that *optimization* of economy will, in fact, necessitate the selection of convoluted "wholistic" encodings in which information is encrypted in the interactions of many features—an

[2]We describe DT-SELECT's tree-building process in greater detail in Section 4, which presents the entire algorithm.

Table 1: Top-level DT-SELECT algorithm.

1. Construct a large pool of potentially useful features
2. Initialize an empty decision tree
3. Initialize training partition to all training examples
4. Select a feature (add a decision tree node):
   - 4a. Score all features in parallel on current partition
   - 4b. Add the most informative feature to the decision tree (or terminate the branch if stopping criterion met)
   - 4c. Subpartition the current training examples according to the values of the chosen feature, if any, and recur (step 4) on each subpartition to build each subtree
5. Output the internal-node features as the new representation

outcome precisely at odds with our desire for transparency. In addition, optimization in the large search spaces of the problems we wish to address is too computationally expensive to be feasible.

Because DT-SELECT searches a very large candidate set containing complex general and domain-specific constructed features, our hypothesis is that the selected features will capture important information about the domain which is not available in a more standard representation. The availability of these features may then enhance the abilities of other inductive learning algorithms to solve the target problem. The use of decision tree construction to select particular sets of features balances the competing desires of coverage, transparency, and economy in a computationally efficient manner.

4 DT-Select

Table 1 gives an outline of the top level of DT-SELECT. We expand on each segment of the algorithm in the following subsections.

4.1 Constructing the Feature Pool

Perhaps the most important component of DT-SELECT is the feature pool, because in it reside all features from which the selection algorithm will choose. It is here that the balancing act between intuitive guidance and large-scale search is performed.

In our current implementation, all features in the pool are Boolean-valued (for efficiency reasons discussed later). Aside from this restriction, they may range anywhere from simple ground-level features from the original dataset to general-purpose relational features to highly complex, domain-specific constructed features. The specific forms of the features is left to the discretion of the user. By designing feature types which one suspects may provide useful information for a problem, one directs the search into areas that appear to be most fruitful

without, however, restricting the search so tightly as to prevent the examination of many different individual features.

To return to our example of recognizing arches, suppose one did not know that the specific comparison between the width and height of a single block was particularly crucial to the problem, yet one suspected that comparisons of block dimensions in general might be important. One could then choose to include in the pool a general class of features of the form, "Is (width/height value U) greater than (width/height value V)," where U and V may each be instantiated with the width or height of any block. Every possible instantiation of this feature type will then be included in the feature pool constructed in step 1 of Table 1. While in this example the number of such features is small and the most important ones are obvious, in general feature instantiations may be much more numerous and feature types much more complex, so that one could not possibly hope to find the best instantiations by hand.

To summarize, the user loosely guides the search by defining feature types, and DT-SELECT then considers all instantiations of these types, allowing the discovery of the most important features from among the many instantiations automatically. In this way we retain a thoroughness of search which is not possible by hand in areas of the search space deemed important while sidestepping the combinatorial problems of a blind search.

We mentioned earlier the fact that all features in the pool are Boolean-valued. The primary reasons for this are space and time efficiency. We wish to be able to search as many features as possible (hundreds of thousands or even millions) so that feature types with many instantiations may be included. However, because constructed features may be arbitrarily complex (depending only on the user's definitions), it is in general expensive to compute the value of a constructed feature for a given example. If we must recompute each feature value every time it is needed during tree building, it becomes impossible to include more than a few thousand features in the pool without incurring prohibitive run times.

The alternative we choose is to precompute and store the value of every feature in the pool for every training example. If there are several thousand training examples and we wish to include at least a few hundred thousand features (a scenario which is not uncommon in difficult real-world problems), this amounts to a substantial number of values which must be stored and accessed efficiently. By using Boolean-valued features, we are able to pack each feature value into a single bit. In addition, the computations needed for tree building can then be accomplished with maximum efficiency through simple logical operations (i.e. AND, XOR) between vectors of these bits as described in later sections. Examples of specific types of features which we implemented for the protein-folding problem are given in the "Experiments" section.

4.2 Building the Decision Tree

Tree building occurs in Table 1's recursive steps 4a–4c. Conceptually, the most informative feature over all the training data is chosen as the root of the tree. The two values of this feature partition the training examples into two subsets, one for which the feature is true and another for which it is false. This process is then repeated recursively for each of the two subsets until a stopping criterion is met. Thus, during each feature selection cycle, all features in the pool are evaluated for their ability to separate the current training partition

into their correct categories (step 4a). This is currently done using Quinlan's *information gain* metric (Quinlan, 1986) modified for multiple categories.[3] Information gain is a method for measuring how well the different values of a feature separate a set of examples into their respective categories; the better the separation, the higher the information gain. The information needed to correctly partition a given set of training examples according to their proper classifications is

$$I(n_1, n_2, ..., n_c) = -\sum_{i=1}^{c} \frac{n_i}{N} log_2 \frac{n_i}{N}$$

where c is the number of categories, n_i is the number of examples in category i, and $N = \sum_{i=1}^{c} n_i$. Then the amount of information gained by partitioning the examples according to the value of a particular feature F is

$$gain(F) = I(n_1, n_2, ..., n_c) - \sum_{j=1}^{v} \frac{\sum_{i=1}^{c} n_{ij}}{N} I(n_{1j}, n_{2j}, ..., n_{cj})$$

where c, n_i, and N are as before and F has v possible values. n_{ij} represents the number of examples of category i having feature value j, so the sum over i in the above formula is simply the total number of examples of all categories for which feature F has value j.

The feature which scores best according to this metric is attached to the decision tree at the point where this particular recursion began (step 4b). Finally, the training examples at this node are partitioned according to the chosen feature's two values (step 4c) and the process is repeated for each such subset which contains examples of more than one class.

This tree-building algorithm is essentially an efficient parallel implementation of ID3 (Quinlan, 1986) except that the features which are used to split the training data are all and only those which reside in the user-defined feature pool. Because all of these are Boolean-valued, the resulting decision trees are binary.

If the only stopping criterion were the complete separation of the training examples into a partition of homogeneous subsets, the tree is likely to overfit the data. To help prevent such overfitting, our current implementation of DT-Select includes a statistical χ^2 test, extended from Quinlan (1986) to multiple categories, which attempts to compare the observed discrimination power of a feature relative to that expected by chance.[4]

If the value of a feature F is independent of the classifications of examples in the current training set (that is, it is an irrelevant feature), then the expected value n'_{ij} of n_{ij} (number of examples of category i having value j for feature F) is

$$n'_{ij} = n_i \left(\frac{\sum_{i=1}^{c} n_{ij}}{N} \right)$$

This is just the number of examples of category i times the proportion of all examples which have value j for feature F. Since F is irrelevant to example classification, we expect each

[3] We also intend to explore the use of Fayyad and Irani's (1992) *orthogonality measure,* which exhibits better performance on several tasks, as a substitute for information gain.

[4] We intend to replace this criterion with more a sophisticated overfitting prevention technique, such as the tree pruning methodology of C4.5 (Quinlan, 1992), in the near future.

individual category to have the same proportion of examples with value j for F as the total set of examples does.

Given the expected values, we can compute the familiar X^2 goodness-of-fit statistic (e.g. Fienberg, 1980)

$$X^2 = \sum_{i=1}^{c} \sum_{j=1}^{v} \frac{(n_{ij} - n'_{ij})^2}{n'_{ij}}$$

which is approximately χ^2 with $(c-1) \times (v-1)$ degrees of freedom. Thus, if the integral of the appropriate χ^2 distribution from 0 to X^2 is > 0.95, the feature discriminates better than chance at the 0.05 confidence level according to this test. If on a given recursion no features in the pool exhibit a discriminatory power which exceeds a user-specified confidence level, feature selection along that branch of the tree is terminated. The current node in the tree then becomes a leaf which is labeled with the most common class of the training examples in its partition.

4.3 The Parallel Implementation

In order to deal with large numbers of features and training examples, we have implemented the feature scoring portion of our tree-building algorithm (step 4a in Table 1) on the Wisconsin Thinking Machines CM-5, a message-passing Multiple-Instruction Multiple-Data machine. This gains us additional processor cycles as well as the memory needed to store vast numbers of precomputed feature values, both of which are necessary to apply this algorithm to real-world problems in reasonable time. This subsection describes the details of the parallel implementation.

At each cycle of tree-building, the program must find the feature of highest information gain which passes the χ^2 confidence test. Thus, these values must be computed for every feature in the pool at each selection step. These calculations require only that we know how many examples of each class have the value *true* and how many the value *false*. Because these calculations can be done for each feature in isolation without affecting computations for other features, the problem of finding the most informative feature is perfectly suited to parallelization by giving each processor responsibility for a particular group of features. After precomputing the feature values, the same amount of computation is required to do the evaluation for each feature regardless of its original complexity. Hence, we simply partition the features evenly across processors without regard for the mix of feature types each one gets, and the processing load is perfectly balanced.

We use a host-node processing paradigm for our implementation, in which one program running on a front-end, or *host,* machine acts as an overseer and bookkeeper for another program which runs in parallel on the CM-5's processing *nodes.* Communication occurs only between the host and nodes. Because the communication requirements of our algorithm are extremely low, this paradigm does not produce the communications bottleneck which might occur in more message-intensive programs.

The tree-building inner loop in which the parallelism occurs is shown in Table 2. Each step is described in more detail below.

First, the features in the pool are partitioned evenly across the nodes (step 1 in Table 2). The global pool itself is never constructed explicitly as a single entity. Instead, its features

Table 2: Tree-building inner loop.

1. Determine local features and receive training examples
2. Precompute feature vectors
3. Repeat until host says to stop
   3a. Receive active example vector
   3b. Find best local feature and send to host
   (Host updates decision tree)

have an implicit canonical ordering, and each processing node generates only those whose index, modulo the number of nodes, equals its node address.

Each node receives a copy of all training examples from the host. Then, in parallel, the nodes precompute the values of their features for the training examples (step 2). Each feature has an associated bit vector in which its (Boolean) value for every training example is stored. The values for all examples of a particular class are contiguous. Feature values are stored in the same order as the array of examples, which is constant across host and nodes. Thus, a particular example's value is always in the same bit position in every feature vector in every node processor.

Figure 3 (top) gives a visual presentation of the feature value table of the node processor with address i. One bit vector denotes the current training partition (active examples), while a table of isomorphic vectors stores the precomputed values of each of this node's features for every training example. A given position in any bit vector corresponds to a fixed single example, and these are grouped together by category, beginning on double-word boundaries for access efficiency. (Thus, there may be some unused bits at the end of each category, shown to the right of the dashed vertical lines in the table.)

To denote the current training partition, or *active* examples, a bit vector of the same morphology as the feature vectors is distributed to the nodes by the host (step 3a in Table 2). This is the first vector at the top of Figure 3. Active examples are denoted by one bits and inactive examples by zero bits. Thus, in order for a node to determine the number of active examples of each class which evaluate to *true* for a given feature, it need only perform a logical AND operation between the active example vector and the feature vector and then count the number of remaining one bits in each category. The number of *false* outcomes for each class is just the total number of examples of that class minus the *true* outcomes.

The first vector at the bottom of Figure 3 shows the outcome of such an AND operation between the active example vector and feature number $i + n$ shown at the top of the figure. The one-bits in this result vector indicate all the active examples which had the value *true* for that feature, and hence constitutes one set in the partition that feature defines with respect to the given active examples. The last vector at the bottom of the figure shows the other set in the partition, namely, those active examples with feature value *false*. This vector is found by an XOR operation between the true partition vector and the original actives vector at the top of the figure. The AND operation is performed once for every feature during a feature selection cycle. The XOR, however, is only performed for the best feature, once this has been determined. This is because we only need to know the number of examples of each

Processing Node i's Main Data Structures

	Category 1		Category 2		...	Category c	
Active Examples	11010001	000	1110111	0000	...	01001000	0000
Feature Number	Feature values for each example (categories as in Active Examples vector)						
i	10111001	000	0011001	0000	...	11110110	0000
i + n	01001110	000	1010011	0000	...	00111000	0000
i + 2n	01101001	000	0100110	0000	...	11010110	0000
i + 3n	11001001	000	0111011	0000	...	01001110	0000
	. . .						
i + kn	11111110	000	1111111	0000	...	11011101	0000

The Active Example Partition Produced by Feature (i + n)

(Feat i+n) AND Act (pos partition)	01000000	000	1010011	0000	...	00001000	0000
(pos part) XOR Act (neg partition)	10010001	000	0100100	0000	...	01000000	0000

Figure 3: Top: the main data structures of the node with address i (n here is the number of nodes). Bottom: the partition of the active examples produced by the feature numbered $i + n$.

class in each partition, not their actual identities, to calculate information gain, and these numbers are readily available from the result of the AND.

All nodes may thus process their own features in parallel, computing the needed information gains and χ^2 values with great efficiency to determine the best feature (step 3b in Table 2). Each node sends the information gain of its best feature to the host, which then determines the global most informative feature (if any pass the χ^2 test) and adds it to the tree. Nodes keep track of the result of the AND operation for their best feature, so the node with the winning feature can provide this to the host for distribution as the active example vector for the positive-outcome recursion. The active vector for the negative-outcome recursion is found via an exclusive-OR with the positive-outcome actives vector and the original actives vector.

The majority of node processing time is spent doing logical AND operations and counting the one bits in the results. Double-word loads are used to minimize bit-vector access time. Currently, the bit counting is implemented using a 16-bit table lookup, where the table stores the number of one bits in every possible 16-bit number. Other methods are possible. For example, one bits could be counted by repeated shifts and comparisons to see if the high bit is set. This increases the number of operations but decreases the number of memory accesses. In a similar vein, a smaller 8-bit lookup table, though it requires twice the number of operations as the 16-bit lookup, could conceivably exhibit better cache performance than the larger table. In practice, the 16-bit lookup appears to be slightly faster than the 8-bit lookup. At this time we have not done performance analyses to compare table lookup to shifts and compares.[5]

It is of note in this implementation that, because feature values are stored in bit vectors of constant morphology, the values of every feature for every training example are accessed at each feature selection cycle (Figure 2's step 3b), even though only some of the examples are active. This means each node of a decision tree requires constant time to produce. A more typical tree-building algorithm would recompute feature values as needed and thus only look at the values of the active examples. Since examples are partitioned across each level of a decision tree, this requires only constant time per *level*. However, in general it is orders of magnitude more expensive to compute individual feature values than to perform register operations on precomputed bit vectors. This leads to a very large constant for the typical approach, whereas the constant in our implementation is small. Early experience with serial prototypes indicated that a more standard approach requiring recomputation would be too slow for the trees we typically build in practice. In addition, tree growth is limited in our implementation by the χ^2 test, thus preventing the constructions of extremely branchy trees which might begin to exhibit combinatorial blow-up. In short, even though the theoretical growth rate of our implementation is worse for dense trees than that of more familiar implementations, in practice we currently stay well below the point at which the two curves cross each other.

[5] A fourth possibility is to do some assembly-language programming in order to make use of a bit-counting instruction available on the CM-5's accompanying vector processing units.

Table 3: An example of a possible protein fragment. The top line gives the primary sequence as single-character amino acid abbreviations. The bottom line classifies each AA into the type of secondary structure in which it participates: "a" = α helix, "b" = β strand, and "." = random coil.

Input:	NRQYGGGRHMGRQYNVVTQESSPSACADNVGINGGSLAENTTKFLNLKDSNY
Output:	....aaaaaaaa.....aaaaaaaaaaaaaa......bbbbb..........

4.4 The Selected Feature Set

The final step of DT-SELECT (step 5 in Table 1) is to output the features of the internal nodes of the decision tree. This set of features serves as the representation selected by the algorithm. It may seem odd that we inductively build a decision tree to select features for other inductive learning algorithms. After all, a decision tree is a classifier in itself, so why not just stop there? The fact is, it is entirely possible to do so. However, in our experiments with predicting protein secondary structure, detailed in the following sections, we have found that the decision trees themselves do not perform well on this task. In addition, we are interested in the general feature-selection problem itself, and several advantages of using decision trees to accomplish this have already been discussed.

5 Protein Secondary-Structure Prediction

Our first test of the DT-SELECT approach involved the task of predicting protein secondary structures.[6] This is a problem from molecular biology in which one attempts to predict the high-level shapes, or *secondary structures,* which globular protein molecules will fold into when in solution. Proteins are large molecules in the form of linear chains of amino acids (AAs; see Table 3), and there are 20 common AAs generally found in protein sequences. A given protein's AA sequence can be determined with relative ease in modern laboratories. However, a much more important factor in a globular protein's functioning is its three-dimensional shape when in solution, which determines the structures and configurations of binding sites. Thus, it is the three-dimensional structure, called *tertiary structure,* which biologists would like to find. This includes the coordinates, orientations, and secondary bonds of each amino acid molecule in space. Knowing the tertiary structure is a main step in understanding how a protein works and therefore is important in such areas as determining protein functional similarity, combating genetic diseases which involve the production of defective proteins, and designing drugs to produce specific desired effects.

While the primary structure is believed to fully determine the tertiary structure, the problem of mapping from primary structures directly to tertiary structures is a very difficult one because interactions on an atomic level are both significant and complex. Thus much work has instead concentrated on predicting the more abstract shapes of a protein's sec-

[6]Sections 5–refDiscussion constitute an expanded presentation of work previously reported in Cherkauer and Shavlik (1993).

ondary structure as an intermediate step on the way to this goal. The most commonly used secondary-structure elements are denoted α *helix,* β *strand,* and *random coil* (Table 3).

An α helix is a regular spring-shaped structure formed by a section of a protein's amino acid chain. There are 3.6 AAs in one turn of an α helix. The number of AAs involved in a complete α helix varies widely from about four to over 40, with the average being about ten (Branden & Tooze, 1991).

A β strand is a linear (i.e. roughly straight) structure formed by usually five to ten consecutive AAs (Branden & Tooze, 1991). β strands do not occur singly; instead, several such strands must align themselves next to each other to form a stable, relatively planar structure. The entire configuration is called a β *sheet.*

Finally, all structures which are not either α helices or β strands are put into the catch-all category of random coil.[7] Branden and Tooze (1991) provide a good introduction to protein structure for the reader who wishes to learn further details.

The problem of predicting protein secondary structures is one that is the subject of much research. For quite some time, researchers in both molecular biology and computer science have attempted to develop rules or algorithms which can accurately predict these structures (e.g. Lim, 1974a, 1974b; Chou & Fasman, 1978; Qian & Sejnowski, 1988; Zhang *et al.,* 1992). Researchers often make use of *inductive learning* techniques, whereby a system is trained with a set of sample proteins of known conformation and then uses what it has learned to predict the secondary structures of previously unseen proteins. However, the form in which the examples are represented is an issue which is often not well addressed. As we have already noted, the performance of inductive learning algorithms is intimately tied to the representation chosen to describe the examples, yet in most cases the learning systems are given only the names of the amino acids in a segment of protein with which to make their predictions. There is a wealth of other information available about the properties of individual amino acids (e.g. see Kidera *et al.,* 1985; Hunter, 1991) which is ignored by these representations. The experiments reported in the following sections test the hypothesis that inclusion of this information should improve the predictive accuracy of inductive algorithms for secondary-structure prediction.

6 Features for Protein Secondary-Structure Prediction

Because of the complexity of the secondary-structure prediction task, it is difficult to know what features, from the myriad of possibilities, might be the most propitious for learning. The biological literature provides some clues to the kinds of features which are important for this problem (Lim, 1974a, 1974b; Chou & Fasman, 1978; Kidera *et al.,* 1985); it is entirely possible, however, that different combinations or variations of these may also prove valuable for the learning task. Humans cannot be expected to analyze by hand extensive numbers of such features, yet this kind of search may yield valuable fruit for such a challenging problem. Thus, this problem is a perfect testbed for the DT-SELECT approach. The subsections that follow describe the features which comprised the pools for our protein folding experiments. (The individual pools are summarized in Table 4.)

[7]Secondary structures other than α helix, β strand, and random coil are occasionally defined, but these are the structures we deal with in the current experiments.

Table 4: Feature pools for Tree 1 through Tree 4.

	Tree 1	**Tree 2**	**Tree 3**	**Tree 4**
Nominal	√	√	√	√
Unary Numeric	√	√	√	√
Binary Numeric	√	√	√	√
Unary Cluster	√	√	√	√
Binary Cluster	√	√	√	√
Unary Average			√	√
Template	All but • • • • • • • •	All but • • • • • • • •	All	All
Thresholds	$\frac{1}{4}$	$0\ \frac{1}{8}\ \frac{1}{4}\ \frac{1}{2}\ 1$	$\frac{1}{4}$	$0\ \frac{1}{8}\ \frac{1}{4}\ \frac{1}{2}\ 1$
Total Features	**60,990**	**208,290**	**64,890**	**227,790**

Table 5: Partitionings of amino acids according to high-level attributes. Duplicate partitions are given identical numbers.

Structural partition

1. Ambivalent {A C G P S T W Y}
2. External {D E H K N Q R}
3. Internal {F I L M V}

Chemical partition

4. Acidic {D E}
5. Aliphatic {A G I L V}
6. Amide {N Q}
7. Aromatic {F W Y}
8. Basic {H K R}
9. Hydroxyl {S T}
10. Imino {P}
11. Sulfur {C M}

Functional partition

4. Acidic {D E}
8. Basic {H K R}
12. Hydrophobic non-polar {A F I L M P V W}
13. Polar uncharged {C G N Q S T Y}

Charge partition

4. Acidic {D E}
8. Basic {H K R}
14. Neutral {A C F G I L M N P Q S T V W Y}

Hydrophobic partition

12. Hydrophobic {A F I L M P V W}
15. Hydrophilic {C D E G H K N Q R S T Y}

6.1 The Raw Representation

In order to detail the types of features the pool includes, first we must briefly describe the raw example representation from which they are constructed. The proteins we use are those of Zhang *et al.* (1992) and are provided as primary sequences of amino acids (AAs). There are 113 protein subunits derived from 107 proteins in this dataset, and each subunit has less than 50% homology with any other. Each position in a sequence is classified as either α helix, β strand, or random coil. For all of the experiments reported here, each possible 15-AA subwindow of a protein constitutes an *example*, for a total of 19,861 examples (one per residue). The overall task is to learn to correctly predict the classifications of the center AA of unseen examples, given a set of classified examples for training. Subwindow sections which overhang the end of a sequence are filled in with a code denoting complete ambiguity. The same code is also occasionally present within the dataset's proteins themselves; since some of the primary sequences are actually protein fragments, there may be more AAs following the end of a sequence as given, so it is reasonable to represent these areas using the same code as for internal ambiguity. This follows the representation of Zhang *et al.* (1992).

DT-SELECT has available the raw primary representation of each example. In addition, in order to tap AA physical and chemical property information which is not available from the primary representation alone, the implementation also has access to two other sources of information about amino acids. The first is a table of ten statistical factors from Kidera *et al.* (1985). These are small floating point numbers (ranging from -2.33 to 2.41), different for each amino acid, which summarize 86% of the variance of 188 physical AA properties. Since the value of each factor averages to zero across amino acids, we simply use zero as the value of all factors for unknown AAs.

The second collection of AA information is the knowledge of various partitionings of the set of amino acids into groups which share common higher-level aspects. These are shown in Table 5. Though the figure shows 20 labeled subgroups of AAs, there are only fifteen unique partitions, and it is membership in these which the program uses. (The partitions are numbered in the figure such that duplicate partitions share the same number.)

6.2 Constructed Features

All features in the feature pool are constructed from the example information specified in the preceding section. Currently we have implemented several types of general-purpose and domain-specific features, some of which are quite complex. As mentioned earlier, all features have Boolean values.

The types of implemented features are summarized in Table 6. Of these, nominal, unary numeric, and binary numeric comparisons may all be considered general-purpose. Templates and unary and binary clusters are specific to the domain. Average unary features may be viewed as both: though they were inspired by the ideas of Chou and Fasman (1978), they could easily be applied to any domain whose examples contain numeric attributes for which averaging makes sense. Each particular feature type is described in more detail in the following subsections.

Table 6: A brief summary of feature types. See Sections 6.2.1–6.2.7 for further details.

Name	Domain
Nominal	Single amino acid names
Unary Numeric	Single-factor comparisons
Binary Numeric	Two-factor comparisons
Unary Average	Single-factor average comparisons
Template	Salient multi-factor comparisons
Unary Cluster	Single amino acid group memberships
Binary Cluster	Paired amino acid group memberships

6.2.1 Nominal Features

This type of feature asks, "Does example position P contain amino acid A?" Since there are twenty AAs and one ambiguity code in our data, for 15-AA examples there are

$$21\ AAs \times 15\ positions = 315$$

nominal features. We note that these are often the only features a typical artificial neural network is given as inputs for this problem.

6.2.2 Unary Numeric Features

These features compare a particular statistical factor of the amino acid at a given example position with the neutral value of zero. While this is technically a binary comparison, we call these "unary numeric" features because only one statistical factor is involved. The possible comparisons are less-than, equal-to, and greater-than, all of which are performed relative to one of a set of user-specified positive thresholds. A factor is *greater than zero with respect to a threshold* if it is strictly greater than the threshold. Likewise, it is less than zero w.r.t. the threshold if it is strictly less than the negated threshold. Otherwise it is considered equal to zero. A typical set of thresholds is {0.0 0.125 0.25 0.5 1.0}.

An individual unary numeric feature asks, "Is factor F of the AA in example position P { less-than | equal-to | greater-than} zero w.r.t. threshold T?" where the comparison is one of the three possible ones. Thus, using five threshold values there are

$$10\ factors \times 15\ positions \times 3\ comparisons \times 5\ thresholds = 2,250$$

potential unary numeric features. Calculations in the following subsections also assume five threshold values.

6.2.3 Binary Numeric Features

Binary numeric features are similar to unary numeric ones except that they compare statistical factors of amino acids in two example positions with each other. A feature of this type

is of the form, "Is factor F_1 of the AA in position P_1 { less-than | equal-to | greater-than} the factor F_2 of the AA in position P_2 w.r.t. threshold T?" F_1 may be the same as F_2 as long as P_1 and P_2 are different. Similarly, P_1 and P_2 may be the same if F_1 and F_2 are different. Thus, there are

$$\{ ((10 \times 10)\ ordered\ factor\ pairs \times \binom{15}{2}\ unique\ position\ pairs)\ when\ P_1 \neq P_2$$
$$+ (\binom{10}{2}\ unique\ factor\ pairs^{8} \times 15\ positions)\ when\ P_1 = P_2,\ F_1 \neq F_2 \}$$
$$\times\ 3\ comparisons \times 5\ thresholds$$
$$= 167,625$$

possible binary numeric features.

6.2.4 Unary Average Features

These are identical to unary numeric features except that instead of comparing a single statistical factor to zero, they compare the average of a factor over a given subwindow of an example to zero. To wit, these features ask, "Is the average value of factor F of the subwindow with leftmost position P and width W { less-than | equal-to | greater-than} zero w.r.t threshold T?" Subwindows from width two to the full example width of 15 are allowed and can be placed in any position in which they fit fully within the example. Thus, there are 14 subwindows of width two, 13 of width three, ..., and only one of width 15. Summing, there are

$$10\ factors \times \sum_{i=1}^{14} i\ subwindows \times 3\ comparisons \times 5\ thresholds = 15,750$$

possible unary average features.

6.2.5 Template Features

Template features are the first wholly domain-specific features we describe. They essentially represent particular conjunctions of unary numeric features as single features. A template feature picks out several example positions and performs the same unary numeric comparison with the same factor and threshold for all of them. The feature has value *true* if the test succeeds for all positions and *false* otherwise. Currently, three- and four-place templates are implemented. Because template features consolidate the information from several unary numeric features into one, they may have larger information gains than any of the unary numeric features would individually. This is important, because the patterns of AAs examined are restricted to ones which appear to operate in conjunction during protein folding, as explained in the next paragraph.

The three or four positions of a template feature are chosen such that they lie in one of a few user-specified spatial relations to one another (hence the name "template"). For

[8]The set of possible comparisons itself provides symmetry, alleviating the need to include *ordered* factor pairs in this case.

Table 7: Three- and four-place templates used.

Three-place templates	
● ● ●	Local interactions
● ○ ● ○ ●	β-strands
● ● ○ ○ ●	α-helices (hydrophobic triple)
● ○ ○ ● ●	α-helices (hydrophobic triple)
● ○ ○ ● ○ ○ ●	α-helices
● ○ ○ ● ○ ○ ○ ●	α-helices
● ○ ○ ○ ● ○ ○ ●	α-helices
● ○ ○ ○ ● ○ ○ ○ ●	α-helices (hydrophobic run)
Four-place templates	
● ● ● ●	Local interactions
● ● ○ ○ ● ●	α-helices (overlapping 1-5 pairs)
● ○ ● ○ ● ○ ●	β-strands (alternating), α-helices (overlapping 1-5 pairs)
● ○ ○ ● ● ○ ○ ●	α-helices (overlapping 1-5 pairs)
● ○ ○ ○ ● ○ ○ ○ ● ○ ○ ○ ●	α-helices (long hydrophobic run)

example, triplets of amino acids which are four apart in the sequence may be important for α helix formation, since they will all be on approximately the same "face" of the helix. Thus, the user may choose to specify a three-place template of the form "(1 5 9)." We represent this graphically as "(●○○○●○○○●)." This notation defines a spatial relationship, or *template*, among three amino acids; it is not meant to restrict a template feature to looking only at the first, fifth, and ninth amino acid in an example. On the contrary, the template may be "slid" to any position on an example, provided the entire template fits within the example. Thus, this particular template would also generate features which examined the second, sixth, and tenth AAs, as well as ones which accessed the seventh, eleventh, and fifteenth.

To summarize, a template feature asks, "Is factor F {less-than | equal-to | greater-than} zero w.r.t. threshold T for *all* AA's in template M with its left end at position P?" Table 7 gives graphic representations of all the templates used in these experiments, along with annotations as to their expected areas of value. Most of them were suggested by Lim (1974b, 1974b). Using the set of templates in Table 7, there are a total of

$$10\ factors \times 120\ template\ positions \times 3\ comparisons \times 5\ thresholds = 18,000$$

template features.

6.2.6 Unary Cluster Features

These are domain-specific features similar to nominal features, but instead of directly using the names of AAs, they check for membership in one of the groups, or "clusters," of related AAs given in Table 6. These features ask, "Does the AA in position P belong to cluster C?" There are

$$15\ \textit{clusters} \times 15\ \textit{positions} = 225$$

unary cluster features.

6.2.7 Binary Cluster Features

Binary cluster features comprise all pairwise conjunctions of unary cluster features which examine different positions. That is, they ask, "Does the AA in position P_1 belong to cluster C_1 and the AA in position P_2 to cluster C_2?" where P_1 and P_2 are distinct. Thus there are

$$(15 \times 15)\ \textit{ordered cluster pairs} \times \left(\begin{array}{c} 15 \\ 2 \end{array} \right) \textit{positions} = 23,625$$

binary cluster features.

7 Experiments

In order to evaluate the worth of feature sets selected by DT-SELECT, we have compared the classification correctnesses of artificial neural networks (ANNs) which use a standard input representation to those of ANNs whose representations are augmented with these sets. The "standard" ANN input representation encodes only the particular amino acid sequence present in an example. The representations of *augmented* ANNs add to this the features chosen by DT-SELECT. (Preliminary experiments indicated that, for this particular task, augmentation of the standard feature set results in better performance than simply using the features chosen by DT-SELECT alone, perhaps indicating that the stopping criterion during tree building was too strict to allow full representational coverage.) To the fullest extent possible, all experimental conditions except the different representations (and attendant network topologies) were held constant for the corresponding standard and augmented ANNs compared.

7.1 Cross-Validation

All experiments followed a ten-fold cross-validation (CV) paradigm. It is important to ensure that all the examples from a single protein subunit are either in the training *or* the testing set during all cycles of CV to avoid artificially overestimating correctness through effects of training on the testing set. Thus, the experiments were set up by first separating the 113 subunits into ten separate files. Nine of these are used for training and the tenth for testing in each cycle, so each file is the testing set once. These files were created by first ordering the subunits randomly and then, for each subunit in the list, placing it in the file which at that time contained the shortest total sequence length. This method balances the desires for

a completely random partitioning of the data and the obtainment of files of approximately equal size. The same ten files were used in all experiments.

7.2 The Networks

All networks were feed-forward with one layer of hidden units and full connection between layers. (We ran each cross validation experiment using networks having 5, 10, 20, and 30 hidden units.) We initialized network weights to small random values between -0.5 and 0.5, and we trained the ANNs with backpropagation for 35 epochs. On each epoch, correctness on a tuning set consisting of 10% of the training data was tracked, and the weights from the epoch with the highest tuning-set correctness were used for the final, trained network. The output layers contained three units, one for each of α helix, β strand, and random coil, and the one with highest activation indicated a network's classification of an example.

The standard ANNs used a typical "unary" input encoding of the amino acids in an example (which corresponds to the 315 nominal features of DT-SELECT). Specifically, this encoding uses 315 input units: 21 for each of the 15 example positions. For each position, only one of the 21 units is on (set to 1.0) to indicate which amino acid (or the ambiguity code) is present. The remaining input units are set to 0.0. ANNs using an augmented representation have the 315 inputs of the standard ANNs plus additional binary inputs corresponding to the values of the features chosen by DT-SELECT.

To preserve the cross-validation paradigm employed, augmenting features for each fold of the CV had to be chosen separately using only the data in that fold's training set. Therefore, the ten augmented networks for a particular CV run did not always have identical augmenting features or even the same number of features. This is unavoidable if contamination with information from the testing sets is to be avoided. However, the resulting network size differences were small relative to the overall network sizes.

7.3 Augmentations

To explore the utility of representation augmentation by our method, we performed four separate network augmentation experiments using features chosen by DT-SELECT from differing pools of features with varying stopping criterion strictnesses. For each of these experiments, DT-SELECT built ten decision trees, one from each training set, using a fixed feature pool. For convenience, we will lump the first experiment's ten sets of augmenting features under the label "Tree 1." Likewise, we shall call the other sets "Tree 2," "Tree3," and "Tree 4."

The features in the pools used to build these trees are summarized in Table 4. The sizes of the resulting individual trees are given in Table 8. The differences in average sizes of the four sets of trees are due to varying the strictness of the stopping criterion.

7.4 Performance Notes

The use of the Wisconsin CM-5 was crucial in making these experiments possible, both because of its large memory and its parallel computing power. Our CM-5 currently has one gigabyte of main memory for each of two independent 32-node partitions. The largest

of the experiments reported here required approximately 586 megabytes of this. Since the precomputed feature values occupy most of this space, the mere availability of so much real (as opposed to virtual) memory alone adds substantially to tree-building performance by eliminating paging.

Table 8: Average tree sizes (number of features) for Tree 1–Tree 4.

Decision Tree	Avg. Size
Tree 1	31.8
Tree 2	46.7
Tree 3	17.5
Tree 4	32.5

Table 9: Best test-set percent correctnesses (averaged over the ten folds) of the standard and augmented ANNs.

ANN Type	% Correct
Standard	61.5
Tree 1	61.6
Tree 2	61.2
Tree 3	61.9
Tree 4	61.0

We were able to run all of the tree-building experiments in only a few hours of total CPU time on the CM-5. The longest-running (affected by both number of features and strictness of stopping criterion) of the four cross-validated tree-building experiments was Tree 2, which took approximately 3.35 hours on the CM-5 to build and test all ten trees. These trees include a total of 467 features (internal nodes) and 477 leaves.[9] Since each of these required one feature selection cycle (step 4 in Table 1), this implies a total of 944 such cycles for the complete run. (A leaf requires a cycle to determine that no features perform well enough to merit inclusion in the tree at that point.) Since the Tree 2 feature pool contained 208,290 features, and each tree was built from roughly 17,875 training examples, each cycle accessed roughly 3.72×10^9 feature values, for a total of about 3.51×10^{12} feature value accesses over all 944 cycles. This was accomplished in approximately 12,060 seconds using a 32-processing node CM-5 partition, yielding an aggregate search rate in the neighborhood of 2.91×10^8 (or 291 million) feature values per second. (The time needed to construct each node is constant, so about 12.8 seconds were used per node.)[10] Note that the time taken also includes overhead in reading and distributing the examples, precomputing feature values, and evaluating tree performance on the testing sets, so the actual search rate is somewhat higher.

7.5 Results

The best test-set correctnesses (averaged over the ten folds) observed across the four different numbers of hidden units for the standard and augmented networks are given in Table 9 (There was little variation in correctness over the differing numbers of hidden units.) We see that the augmented networks unfortunately did not produce the performance gains we had anticipated achieving with DT-SELECT. The Tree 1 and Tree 3 networks did obtain

[9] A binary tree with n internal nodes always has $n+1$ leaves, regardless of configuration. Since there were ten trees, the number of leaves is just ten more than the total number of internal nodes.

[10] System software development for the CM-5 is currently ongoing. These performance analyses should be taken as rough estimates only and may vary with different versions of the system software.

Table 10: Average correctnesses of the decision trees on the test sets.

Tree 1	Tree 2	Tree 3	Tree 4
55.9%	56.7	57.2	57.6

slightly higher correctnesses than the best standard network, however the improvements are not statistically significant.

It is of some interest to see how well the original sets of decision trees themselves classify the data. This information is given in Table 10. It is evident that the neural networks do substantially better for this problem than the decision trees by themselves.

8 Discussion

It was surprising to us to find that the addition of apparantly salient domain-specific features of great sophistication to the input representations of ANNs does not lead to gains in classification performance. The reasons for this remain unclear at this time, although several possibilities exist.

First, it could be that the ANNs themselves are capable of deriving similar types of features on their own using their hidden units. If this is true, it demonstrates that the power of backpropagation to develop useful internal representations is indeed substantial, given the complexity of the features available to DT-SELECT during tree construction. This implies that perhaps the implementation of even more complex types of salient features which backpropagation is *not* capable of deriving itself is necessary for the use of DT-SELECT to yield performance improvements.

Second, it is possible that we have not yet implemented the best types of features for augmentation. There are always new types of features one can think of adding to the system; for instance, binary average features which examine two factors in two subwindows, or templates which average factors over the AAs they examine. Indeed, we have observed that, in general, adding new feature types tailored to the secondary-structure prediction task, such as the cluster and average features, tends to displace the more general types of features from decision trees out of all proportion to their numbers. In Tree 2, the cluster-type features account for approximately one third of all features chosen, though they constitute only about 10% of the features in the pool. Even more remarkable, the addition of the unary average feature in Tree 3 and Tree 4 resulted in more than half of the features of each of these tree sets being of this type, though they comprise only about one quarter of the features in the pool for Tree 3 and about 7% of those in Tree 4's pool. This indicates that these features are higher in information "density" than the more general-purpose features they displace, yet, strangely, we do not see great advances in either decision-tree or augmented-network correctnesses when such features are added to the pool.

Thus, a third possible explanation for the lack of observed gain is that, for ANNs of this form, the 315 inputs of the standard unary encoding actually capture in a workable form all the information such networks are capable of using. This is an interesting hypothesis in itself, and the reasons behind it, if it is true, would be worth uncovering—especially

considering the effects of input representation on DNA coding-region prediction mentioned earlier (Uberbacher & Mural, 1991; Farber *et al.* 1992; Craven & Shavlik, 1993a, 1993b). Our standard ANNs were used as controls, so the learning parameters were not extensively optimized for maximal performance. (The augmented networks naturally used the same parameters as the standard ones.) However, though the complete system of Zhang *et al.* (1992) attained a considerably higher correctness, the ANN component of their system alone achieved an accuracy only slightly higher than our ANNs did. Some researchers postulate an upper bound of about 80% correctness on this problem using only local information, but it is possible that techniques more sophisticated than feed-forward ANNs are needed to get beyond the low-60% range when using datasets of the size currently available.

9 Future Work

Our current work includes the addition of other protein-specific features to our implementation, the replacement of the χ^2 stopping criterion with a more modern pruning methodology (Quinlan, 1992), and the investigation of a possible improvement over the information gain criterion for feature selection (Fayyad & Irani, 1992). We are also beginning to test DT-SELECT on two other problems: DNA coding-region prediction and handwritten character recognition.

A more general issue we intend to explore is the use of feature-selection methods other than decision trees. We have performed a few initial experiments with two other algorithms, one which builds Foil-like rules (Quinlan, 1990) to describe the individual example classes and another which applies statistical independence tests to select features that are largely orthogonal, but more work is needed to determine the strengths and weaknesses of different selection approaches.

10 Conclusion

In this chapter we motivated the need for automated methods of choosing example representations for machine learning and identified three criteria an ideal representation should satisfy: coverage, economy, and transparency. We then introduced the DT-SELECT system to address these concerns. DT-SELECT uses decision trees constructed through parallel computation to produce representations which manifest our three desiderata: features are chosen for their transparency; coverage of the training data is increased as features are added to the tree; and the selection of redundant features is discouraged by the characteristics of the information-gain metric and decision-tree structure. The parallel nature of the algorithm allows us to search large candidate pools of features, making it possible to perform automated representation construction for real-world problems.

We tested the utility of our approach by using the sets of features chosen by DT-SELECT to augment the input representations of artificial neural networks that attempt to predict protein secondary structure. Unfortunately, no significant gains in correctness were observed on this difficult problem, leading us to surmise three possible explanations for the negative result:

- The ANNs could derive the features we used on their own
- Additional feature types are needed for performance gains
- The simple, standard encoding is already nearly optimal for ANNs

Regardless of which of these, if any, is correct, we believe our method exhibits potential for extension in this domain and for application to other problems, both inside and outside molecular biology, as a representation-selection technique.

The core of DT-SELECT is essentially a fast parallel implementation of ID3 (Quinlan, 1986) that has access to a set of user-defined, Boolean-valued constructed features. The implementation can thus be used to construct decision trees from large feature spaces as an end in itself. For problems where decision trees work well, the ability to do this large-scale search may lead directly to improved classification performance. However, even for problems where decision trees work poorly, the features they select can be used as representations for more appropriate learning algorithms.

Machine learning is in general a computationally intensive field. In order to tackle real-world problems in a nontrivial manner, we believe it is important to make use of the parallel computing power which is now becoming available. DT-SELECT is our first effort in the exciting direction of massively parallel methods for machine learning. As parallel machines become more common, we hope other researchers will also take advantage of the CPU cycles they offer by developing additional parallel machine learning algorithms, enabling learning methods to be applied to increasingly large and complex problems.

11 Acknowledgements

This work was supported by Department of Energy Grant DE-FG02-91ER61129 and National Science Foundation Grants IRI-9002413 and CDA-9024618.

12 References

Almuallim, H., & Ditterich, T.G. (1991). Learning With Many Irrelevant Features. *Proceedings of the Ninth National Conference on Artificial Intelligence,* Vol. II (pp. 547-552). Anaheim, CA: AAAI Press/The MIT Press.

Blum, A.L., & Rivest, R.L. (1992). Training a 3-Node Neural Network is NP-Complete. *Neural Networks,* 5, 117-127.

Branden, C., & Tooze, J. (1991). *Introduction to Protein Structure.* New York, NY: Garland.

Cherkauer, K.J., & Shavlik, J.W. (1993). Protein Structure Prediction: Selecting Salient Features from Large Candidate Pools. *Proceedings of the First International Conference on Intelligent Systems for Molecular Biology.* Bethesda, MD: AAAI Press.

Chou, P.Y., & Fasman, G.D. (1978). Prediction of the Secondary Structure of Proteins from their Amino Acid Sequence. *Advances in Enzymology,* 47, 45-148.

Craven, M.W., & Shavlik, J.W. (1993a). Learning to Predict Reading Frames in *E. coli* DNA Sequences. *Proceedings of the Twenty-sixth Hawaii International Conference on System Science* (pp. 773-782). Maui, HI: IEEE Computer Society Press.

Craven, M.W., & Shavlik, J.W. (1993b). Learning to Represent Codons: A Challenge Problem for Constructive Induction. *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence.* Chamberry, France: Morgan Kaufmann.

Farber, R., Lapedes, A., & Sirotkin, K. (1992). Determination of Eukaryotic Protein Coding Regions Using Neural Networks and Information Theory. *Journal of Molecular Biology,* 226, 471-479.

Fayyad, U.M., & Irani, K.B. (1992). The Attribute Selection Problem in Decision Tree Generation. *Proceedings of the Tenth National Conference on Artificial Intelligence,* (pp. 104-110). San Jose, CA: AAAI Press/The MIT Press.

Fienberg, S.E. (1980). *The Analysis of Cross-Classified Categorical Data, Second Edition.* Cambridge, MA, and London, England: MIT Press.

Hunter, L. (1991). Representing Amino Acids with Bitstrings. *Working Notes, AAAI Workshop: AI Approaches to Classification and Pattern Recognition in Molecular Biology,* (pp. 110-117). Anaheim, CA.

Kidera, A., Konishi, Y., Oka, M., Ooi, T., & Scheraga, H.A. (1985). Statistical Analysis of the Physical Properties of the 20 Naturally Occurring Amino Acids. *Journal of Protein Chemistry,* 4, 1, 23-55.

Lim, V.I. (1974a). Algorithms for Prediction of α-Helical and β-Structural Regions in Globular Proteins. *Journal of Molecular Biology,* 88, 873-894.

Lim, V.I. (1974b). Structural Principles of the Globular Organization of Protein Chains. A Stereochemical Theory of Globular Protein Secondary Structure. *Journal of Molecular Biology,* 88, 857-872.

Matheus, C.J., & Rendell, L.A. (1989). Constructive Induction on Decision Trees. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence,* (pp. 645-650). Detroit, MI: Morgan Kaufmann.

Michalski, R.S. (1983). A Theory and Methodology of Inductive Learning. *Artificial Intelligence,* 20, 111-161.

Pagallo, G. & Haussler, D. (1990). Boolean Feature Discovery in Empirical Learning. *Machine Learning,* 5, 71-99.

Qian, N., & Sejnowski, T.J. (1988). Predicting the Secondary Structure of Globular Proteins Using Neural Network Models. *Journal of Molecular Biology,* 202, 865-884.

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

Quinlan, J.R. (1990). Learning Logical Definitions from Relations. *Machine Learning,* 5, 239-166.

Quinlan, J.R. (1992). *C4.5: Programs for Machine Learning.* San Mateo, CA: Morgan Kaufmann.

Ragavan, H., & Rendell, L.A. (1993). Lookahead Feature Construction for Learning Hard Concepts. *Proceedings of the Tenth International Machine Learning Conference.* Amherst, MA: Morgan Kaufmann.

Uberbacher, E.C., & Mural, R.J. (1991). Locating Protein Coding Regions in Human DNA Sequences by a Multiple Sensor–Neural Network Approach. *Proceedings of the National Academy of Sciences (USA),* 88, 11261-11265.

Zhang, X., J.P. Mesirov, D.L. Waltz (1992). A Hybrid System for Protein Secondary Structure Prediction. *Journal of Molecular Biology,* 225, 1049-1063.