Learning Structural Decision Trees from Examples*

Larry Watanabe
Beckman Institute and
Dept. of Computer Science
University of Illinois
Urbana, IL 61801

Larry Rendell
Beckman Institute and
Dept. of Computer Science
University of Illinois
Urbana, IL 61801

Abstract

STRUCT is a system that learns structural decision trees from positive and negative examples. The algorithm uses a modification of Pagallo and Haussler's FRINGE algorithm to construct new features in a first-order representation. Experiments compare the effects of different hypothesis evaluation strategies, domain representation, and feature construction. STRUCT is also compared with Quinlan's FOIL on two domains. The results show that a modified FRINGE algorithm improves accuracy, but that it is sensitive to the distribution of the examples.

1 Introduction

Structural learning, also known as relational learning, has a long if not prominent history in machine learning. Like attribute-value concept learners, structural concept learners try to induce a description of a concept from positive and negative examples of the concept. Structural concept learners differ in using a more expressive first-order predicate calculus representation for their hypotheses.

One of the reasons for the unpopularity of structural learning is the problems posed by learning in a first-order representation. Examples are described in terms of objects and the relationships among them; hypotheses may include existentially quantified variables. The ability to quantify over objects adds a great deal of expressive power to the representation. However, even matching a hypothesis to an example is an NP-complete problem [Haussler, 1989].

Recently, the advantages of a first-order representation have motivated further research into learning in this hypothesis language. FOIL [Quinlan, 1990a] KATE [Manago, 1989], are some recent systems that can learn structural concepts.

This paper introduces a system, STRUCT, that learns structural concepts from positive and negative examples. STRUCT integrates and extends previous work in structural and attribute-value machine learning research. From KATE it borrows the use of decision trees to learn structural concepts. From FOIL it borrows the approach and several techniques. From FRINGE [Pagallo and Haussler, 1990] it borrows the iterative, adaptive feature construction algorithm to produce more concise, accurate, decision trees. From INDUCE [Dietterich and Michalski, 1981] it borrows the representation of examples.

The main purpose of this paper is to explore how these different techniques can be integrated into a structural concept learning system. In addition, this paper describes the results of empirically evaluating several of these refinements.

The paper is organized as follows. First, we describe related research in section 2. The algorithm is described in section 3. Next, we describe some experiments in structural domains and discuss the results of these experiments. Section 5 presents the conclusions of this research.

2 Related Work

In this section, we review systems that use an information-theoretic approach to learning disjunctive structural descriptions. There are many systems that use other approaches, or address different problems in structural learning. Some of these systems are described in [Kodratoff and Ganascia, 1986; Iba *et al.*, 1988; Muggleton, 1990].

2.1 Separate-and-conquer

Separate-and-conquer algorithms for structural domains, also known as covering algorithms, try to find a DNF description of a concept by iteratively generating disjuncts that cover some of the positive examples while excluding the negative examples.

An early application of this algorithm to structural domains is INDUCE [Dietterich and Michalski, 1981]. INDUCE generates disjuncts by constructing a cover from from the literals of one positive example, called the seed.

Quinlan's more recent FOIL [1989; 1990] differs from INDUCE in a number of ways. First, it uses a tuple-based learning paradigm, as opposed to an example-based learning paradigm. Second, FOIL does not restrict the cover to use literals from a seed but searches

*This research was supported in part by NSF grant IR1 8822031.

the entire space of literals. We refer to this as full-width search, because all one-step specializations are considered as candidates. However, having evaluated the candidates, FOIL commits itself to one particular choice and discards the other candidates. In contrast, INDUCE chooses a subset of the candidates for specialization. Third, INDUCE uses a general user definable evaluation function, whereas FOIL's information-theoretic evaluation function is an integral part of the system.

Another recent separate-and-conquer algorithm for learning structural descriptions is Pazzani and Kibler's [1990] FOCL algorithm. FOCL uses a form of typed logic with sortal and semantic constraints to reduce the number of literals considered for inclusion in the cover. This improves efficiency and accuracy. FOCL can also use incomplete and incorrect background knowledge to aid induction, and an iterative-widening search to find a cover for the examples.

2.2 Divide-and-conquer

Although divide-and-conquer algorithms, also known as decision tree inducers, have been widely used in attribute-value domains, until recently few systems have used this strategy in structural domains. Bergadano and Giordana's [1988] ML-SMART system uses a specialization tree approach that uses heuristics and domain knowledge to drive induction. Manago's [1989] KATE is a decision-tree algorithm that uses an object-oriented frame language equivalent to first-order predicate calculus. KATE makes extensive use of the structure provided by the object hierarchy and heuristics to control the generation of literals considered as branch tests. This system has a strong bias against introducing new existentially quantified variables into the description, unlike FOIL which favors introducing new variables. Later, we discuss an experiment comparing the two strategies. Man ago has also used INDUCE's partial-star algorithm as a feature construction algorithm for KATE.

2.3 Adaptive Feature Construction

Like FOIL, Pagallo and Haussler's [1990] FRINGE uses a tuple-based learning paradigm, but unlike the structure-oriented systems, FRINGE uses the tuples simply as examples for atribute-based leaning. FRINGE also differs from INDUCE, FOIL, FOCL, KATE, and ML-SMART in that FRINGE accepts only the training examples, no other knowledge. Nevertheless, FRINGE and its successors Symmetric FRINGE [Pagallo, 1990] and DCFringe [Yang *et* al., 1991] have been shown to improve accuracy. These algorithms learn structure in the form of new features constructed as more and more complex conjunctions and disjunctions of the original attributes. The scheme is to perform iterative feature construction at the leaves of successive decision trees output by an ID3-like algorithm. Extensions of the FRINGE method have been tested by Matheus [1990] which do not restrict constructions to the fringe nodes [also see Yang *et* al., 90].

3 STRUCT

Our system learns structural decision trees from positive and negative examples.

```
Databases:
DB1 =
        father(Christopher, Arthur),
        father(Christopher, Victoria),
        mother(Penelope, Arthur),
        mother(Penelope, Victoria),
        brother(Arthur, Victoria),
        sister(Victoria, Arthur),
        son(Arthur, Christopher),
        son(Arthur, Penelope),
        daughter(Victoria, Christopher),
        daughter(Victoria, Penelope),
        husband(Christopher, Penelope),
        wife (Penelope, Christopher).

Train:
        father(Christophcr, Arthur) :- DBI
        -father(Victoria, Arthur) :- DBI
Classes:
        father(X, Y)
```

Figure 1: Input for STRUCT.

3.1 Representation

STRUCT represents relations as Horn clauses. A Horn clause can be viewed as a logical implication, where the consequent consists of a single literal, called the head of the Horn clause, and the antecedent consists of zero or more literals, called the body of the Horn clause. This representation of examples is similar to the inductive assertions used in INDUCE. In contrast, FOIL learns a relation from positive and negative tuples of a relation. The following shows the input for STRUCT: The *Databases* section defines zero or more dcitabases that can be referenced by examples. Examples are horn clauses, whose antecedent may be an explicit list of literals or a reference to a database. The *Classes* section defines the classes that are to be included in the decision tree. Unifying the head of the training examples with the class literal yields the signed substitutions:

```
+: {(Christopher/X),(Arthur/Y)>
-: {(Victoria/X),(Arthur/Y)}
```

where the sign of the substitution indicates whether one of the literals was negated before unifying. In contrast, FOIL would be given the tuples

```
+: <Christopher,  Arthur>
-: <Victoria,  Arthur>
```

which are isomorphic to the substitutions constructed by STRUCT. The *Classes* may be followed by a *Test* section giving the test set.

The ability of STRUCT to store data in multiple databases rather than one global database can make induction more efficient. Information that is not relevant to an example can be ignored when learning from that example. If this kind of relevance information is not available, all the data can still be stored in a single database.

STRUCT also represents knowledge in the form of Horn clauses. This knowledge may be prior knowledge given to the system or a constructed feature. Before constructing the decision tree, STRUCT computes the

```
Create Tree (class, examples)
    - create a node root labelled by the literal class.
    - select the subset of examples whose head
        unifies with class or —*class
    - RecursiveSplit(root)

RecursivcSplit (node)
    If all training examples at node
        are positive or negative examples,
        label node with pos or neg.
    else
        - select a test based on one literal
        - label node with the literal
        - create child nodes, left and right
        - place the examples that match node at left
            and the others at right
        - Recursivc Split(left)
        - RecursiveSplit (right)
```

Figure 2: Recursive Splitting Algorithm.

deductive closure of the body of each example, and replaces the body with the closure. This form of logical constructive induction has previously been implemented in INDUCE.

3.2 Structural Decision Trees

In this section we describe the decision tree formed by STRUCT and how it is used to classify examples. STRUCT learns a Boolean decision tree as shown in Figure 2. The decision tree algorithm is modified to handle structural descriptions by associating a Horn clause with each of the nodes of the tree, and SLD-resolution is used as the match procedure. The clause associated with a node is defined by the following mapping. Let $node_0, ..., node_n$ be the nodes along the path from the root to node $node_n$. Define L_i, to be the literal labeling node *nodei* if *nodei* is the left child *of node.i-.1* otherwise its negation. Then the clause $L_0 \leftarrow L_1, ..., L_n$ is associated with node $node_n$. The node $node_n$ matches an example if the head of the example can be derived with SLD-resolution from the above clause and the literals in the body of the example.

3.3 Generating Literals for Branch Tests

In this section, we first review FOIL'S method for generating literals for branch tests, then compare its method to STRUCT's method.

FOIL generates the literals used as tests using the following procedure: If the clause associated with the current node is $P(X_1, ..., X_m) \leftarrow L_1, ..., L_n$ then a new literal of form $X_j = X_k, X_j \neq X_k, Q(V_1, ..., V_r)$, or $\neg Q(V_1, ..., V_r)$ can be added to the clause, where Xi's are existing variables, the Vi's are existing or new variables, and *Q* is some relation. The entire space of these literals is searched with the following exceptions:

- literals may be pruned
- the literal must contain at least one existing variable
- the literal must satisfy some constraints designed to avoid problematic recursion.

In contrast, STRUCT generates the literals used as tests using the following procedure: if the current node is *current,* the parent of *current* is *parent,* the clause associated with *parent* is $P(X_1, ..., X_m) \leftarrow L_1, ..., L_n, \theta$ is a substitution that matches the current node to an example, $Q(A_1, ..., A_k)$ is a literal in the example, and $Q(V_1, ..., V_k) \circ \theta = Q(A_1, ..., A_k)$, then $Q(V_1, ..., V_k)$ is a candidate literal for partitioning *current.* STRUCT also considers all candidate literals of form $X_j = X_k$, where Xi's are variables in the clause associated with *parent.* The entire space of these literals is searched with the following exceptions:

- the literal must contain at least one variable from *parent*
- the literal must satisfy constraints to avoid problematic recursion.

Pazzani and Kibler [1990] analyze the complexity of FOIL's approach, and describe how sortal and semantic constraints are incorporated into their system, FOCL. The complexity of FOIL's strategy, without pruning, is approximately $(n + k - 1)$, where n is the number of old variables in the clause, and *k* is the arity of the predicate in the new literal. This must be repeated for each predicate and each generated literal must be matched against the examples, so the cost of generating and choosing a literal for a node is upper bounded by $t \cdot p \cdot (n + k - 1)$, where *t* is the number of tuples covered by the current clause, and *p* is the number of predicates. Quinlan reports that pruning results in a dramatic improvement in the efficiency of the algorithm.

STRUCT's strategy is comparable in complexity to FOIL. The number of substitutions is the same as the number of tuples in Quinlan's formalism. The cost of generating and choosing a literal is upper bounded by $t \cdot l \cdot (v + k - 1)^k$, where *t* is the number of matches of the parent node to the examples, *l* is the number of literals, and *v* is the maximum number of variables that are bound to the the same constant in any substitition.

3.4 Evaluating Literals

STRUCT uses the following evaluation function to evaluate candidate literals. Let *S* be the set of examples at a node, *X* a literal, S_x the subset of *S* that matches X, and $S_{x,y}$ the subset of S_x that belongs to class *y.* Then the evaluation function gives *X* the value:

$$\sum_{x=X,\neg X} \sum_{y=+,-} |S_{x,y}| \log\left(\frac{|S_{x,y}|}{|S_x|}\right)$$

Maximizing this evaluation function is equivalent maximizing information gain [Quinlan, 1983]. FOIL's evaluation function sums over the tuples covered by the new clause, and is therefore asymmetric. The above evaluation function sums over both the matched and unmatched examples (not tuples).

FOIL gives a small credit to a literal that introduces new variables. STRUCT may give either a small reward or penalty $(\pm .1)$ for each new variable in a literal. In our experimental section, we compare the effects of the reward and penalty strategies on the accuracy of the learned concepts.

1. Delete all instances of the relation to be learned from the database.
2. As the decision tree is constructed, whenever a leaf node is created, add the literals in the heads of the examples at the leaf nodes to the database.

Figure 3: Recursive Learning Strategy.

3.5 Recursive Definitions

One issue that faces structural learners is the problem of recursive definitions. Quinlan [1990] proposes a partial ordering strategy that can eliminate some, but not all, of the problematic recursion.

The problem arises because instances of the relation being learned are in the knowledge base. Clearly, the relation itself is the best feature for splitting the positive and negative examples; i.e.

$$P(X_1, ..., X_n) \leftarrow P(X_1, ..., X_n)$$

perfectly splits the tree. But there is an implicit requirement on the learning system that it be able to classify future examples without knowing the same kind of information that is available to it during training, namely the classification of the example.

One possible approach to this problem is to explicitly remove from the database information that the decision tree is supposed to learn, unless the decision tree has already learned it (see Figure 3). STRUCT currently does not implement this procedure, but it avoids adding new literals with the same predicate and variables as the head of the clause. STRUCT also has a non-recursive mode, where definitions are may not add literals with the same predicate as the one being learned.

A second problem that can arise is an infinite regression during induction. For example, suppose the description associated with the current node is Q <— P(X, Y, *Z)* and the literal to be added is $P(X, Y', Z')$. The new definition matches exactly the same set of examples, and the system may continue to add another alphabetic variant of the literal indefinitely:

$$Q \leftarrow P(X, Y, Z), P(X, Y', Z'), P(X, Y'', Z'')...$$

To avoid this problem, STRUCT forms co-designation constraints on the possible bindings of a literal. These arc created for every pair of literals Li, and *Lj,* where

- *Lj* is a new literal that has just been added to the definition
- *Li* is a literal with the same predicate as *Lj.*
- The j-th argument of *Lj* is either a new variable, or is the same variable as the j-th argument of Li,.

The constraint on unification of *Lj* is defined as follows: let $Y_1, ..., Y_n$ be the new variables in *Lj,* and $X_1, ..., X_n$ be the variables in *Li* at the corresponding argument positions. Then $Y_1 \neq X_1, ..., Y_n \neq X_n$ are added as co-designation constraints on unification of *Lj.*

These co-designation constraints prevent the inductive recursion that sometimes occurred in earlier versions of STRUCT.

find-feature1 *{leaf)*
Let L_p be the literal at the parent node of *leaf*
Let L_g be the literal at the grandparent node of *leaf*

If *leaf* is to the left of its parent
$L_0 = L_p$
else
$L_0 = \neg L_p$
If *leaf* is to the left of its grandparent
$L_1 = L_g$
else
$L_1 = \neg L_g$
Return $P(X_1, ..., X_n) \leftarrow L_0, L_1,$
where *Xi* are the variables from the unnegated literals in *LO, L* and *P* is a new predicate.

fmd-feature2(/ea/)
Let L_r be the literal at the root of the tree
Let $P(X_1, ..., X_n) \leftarrow L_0, L_1,$
be the feature returned by find-feature1 *(leaf).*
Return $P(X_1, ..., X_n) \leftarrow L_0, L_1,$
with guard L_r

Figure 4: Feature Construction Algorithm.

3.6 Feature Construction

One problem with decision trees as a representation scheme is that DNF concepts cannot be represented concisely. Representing a DNF concept as a decision tree may require replicating many parts of each disjunct in different branches of the tree. Pagallo and Haussler [1990] call this the replication problem, and propose an adaptive, iterative feature construction algorthm, FRINGE, as a solution to this problem.

Pagallo and Haussler have conducted experiments with FRINGE in random and real-world domains. In the random domain experiments, FRINGE produced more concise and more accurate decision trees for small DNF. In the real-world domain experiments, FRINGE produced decision trees that were at least as accurate and more concise. In STRUCT, we investigated a modification of the FRINGE algorithm for a structural decision tree. The algorithm is essentially the same as FRINGE, with a few changes to handle variables. The algorithm begins with the initial set of relations, and constructs a decision tree as described in the previous sections. Next, a *find-features* procedure generates a new set of relation definitions by examining the fringe of the tree, by calling *find-feature J* for each leaf in the tree at depth $\geq$ 2 (see Figure 4).

We also implemented a variant of *find-features,* motivated by the following considerations. In the original *find-features,* the variables in literals at the fringe of the tree are usually bound by literals higher in the tree. However, the bindings are lost when the fringe literals are made into an independent feature. Thus, the feature tends to be overly general if the literal is unnegated, or overly specific if the literal is negated.

An additional source of potential problems is that the literal in the head of the example contains information that is ignored during feature construction. However, this literal cannot be added to the body of a feature

definition because it matches literals in the head of an example, not the body. Accordingly, we modified the rule representation and matching algorithm to handle *guard* literals. These literals must match the head of an example as a precondition of the rule.

The *find-features* procedure constructs new relation definitions. The new relation definitions are added to the knowledge base, in the form of Horn clauses. Old relation definitions are removed from the knowledge base if they define a relation that is not in the decision tree, and are not necessary for computing a relation that is in the decision tree. This method of feature pruning is Pagallo's *Keep Used Last* method. The examples are replaced by the deductive closure of the base level literals under the new knowledge base. Then, the decision tree algorithm and *find-features* procedure is repeated. The process stops when the same decision tree is constructed as in the previous iteration, or when a fixed number of iterations (7) is exceeded.

4 Experiments

We had several objectives in performing the experiments. First, we wanted to evaluate the overall performance of the system on some standard domains. Second, we wanted to determine if Quinlan's heuristic of assigning a small credit to new variables worked well in a decision tree system. Third, we wanted to investigate the effectiveness of a modified FRINGE strategy for structural domains.

4.1 Kinship Domain

Hinton [1986] implemented a connectionist system for learning family relationships in two families. For a detailed description of the learning task, we refer the reader to Quinlan's excellent [1989, 1990] descriptions.

4.2 Method

In our experiments, we followed the procedure described in [Quinlan, 1990b]. The instances of each relation were partitioned into input-output vectors that have the same relation name and same second argument. Only input-output vectors with at least one positive instance were used. There are 24 instances (equivalently, tuples or examples) in each of the total of 104 input-output vectors.

The 104 input-output vectors were randomly partitioned into training and testing sets. The size of the training set was varied among the values 4, 10, 20, and 40. Each of the test items was scored as correct only if all 24 examples within the vector were correctly classified.

We extended Quinlan's experiment by manipulating whether new variables were credited or penalized in the evaluation of literals. The same samples were used across conditions with the same sample size. Each condition was run 20 times. These results are summarized in Table 1.

4.3 Results and Observations

STRUCT performed significantly better than FOIL on this domain for conditions with the same sample size. Perhaps family relationships, being trees, are easily learned by a decision-tree algorithm.

Table 1: Accuracy Results on Kinship Domain.

Training Items	Testing Items	FOIL	STRUCT (reward)	STRUCT (penalty)
100	4	98 (%)	100 (%)	100 (%)
94	10	90 (%)	99 (%)	99 (%)
84	20	81 (%)	97 (%)	98 (%)
64	40	53 (%)	86 (%)	89 (%)

For STRUCT, assigning a small credit to new variables resulted in decision trees of no better or worse accuracy. Quinlan's rationale for the credit strategy is that adding new variables will allow new literals to be introduced that might have more information gain. This locally increases the size of the hypothesis space. However, the larger hypothesis may also result in overfitting.

4.4 Chess Endgame

The chess endgame domain was used in a comparison of human and machine learning formalisms [Muggleton *et al.* , 1989] Muggleton *et al.* concluded that the ability to produce high-performance in this domain was almost entirely dependent on the ability to express first-order predicate relationships. Thus, this domain seems to be a natural test-bed for structural learning. Quinlan's [1990] results for FOIL on this domain are included in Table 2.

The problem faced by the learner in the chess endgame domain is to learn the concept of an *lilegal* position involving three pieces a white rook, white king, and black king. A position is *Illegal* if either king is in check or two pieces occupy the same square.

4.5 Experiment 1

In this experiment we compared three strategies against a default strategy for training sizes of 100 and 1000. The control condition assigned a penalty for new variables in literal tests, and used a symmetric evaluation function. The other conditions differed from the control condition in one aspect of their strategy. The *reward* condition assigned a small reward for new variables. The *asym* condition used an symmetric evaluation function similar to the one used in FOIL, except that it evaluated information about examples rather than tuples.

Each condition was run with 10,000 testing examples and repeated for 10 trials. The same samples were used across conditions with the same sample size, and the two-tailed correlated t-test was used to test significance.

4.6 Results and Observations

As shown in Table 2, STRUCT achieved comparable accuracy to FOIL for both sample sizes; their mean accuracies are within one standard deviation of each other's. However, FOIL finds descriptions of greater simplicity than STRUCT.

The evaluation function results in Table 3 indicate that on the chess endgame domain, STRUCT performs better with the symmetric evaluation function (the control condition) than with the asymmetric evaluation function. The *sym* condition produced more accu-

Table 2: Accuracy Results on Chess Endgame Domain.

System	Accuracy (%) for 100 Training Objects	Accuracy (%) for 1000 Training Objects
FOIL	92.5 ± 3.6	99.4 ± 0.1
STRUCT	94.6 ± 2.9	99.3 ± 0.2

Table 3: Compairison of Evaluation Functions.

Condition	Train	Accuracy (%)	Confidence
control	100	94.6 ± 2.9	NA
reward	100	93.2 ± 2.7	($p > .005$)
asymm.	100	87.1 ± 2.7	($p > .005$)
control	1000	99.3 ± 0.21	NA
reward	1000	99.3 ± 0.26	NA
asymm.	1000	95.2 ± 0.19	($p > .005$)

rate trees, particularly for larger sample sizes. Perhaps the symmetric evaluation function is appropriate for the symmetric decision tree algorithm, and the asymmetric evaluation function is apppropriate for the asymmetric separate-and-conquer algorithm.

On this domain, assigning a small credit to new variables was also significantly worse than assigning a small penalty to new variables. This result is not surprising as a complete and consistent description can be constructed without introducing any new variables.

Feature construction did not result in a significant increase or decrease in accuracy in any condition, so these results are not included in Table 3. For this data, feature construction's only advantage is the greater conciseness of the decision tree. Our results are consistent with the results in [Matheus, 1990; Pagallo and Haussler, 1990] which showed little or no improvement in accuracy for the datasets obtained from the UCI Repository of Machine Learning Domains. Matheus conjectures that the poor results are due to the high quality of the initial features.

4.7 Experiment 2

This experiment was motivated by the disparity in Pagallo's [1990] results for feature construction on synthetic domains and the UCI domains. We conjectured that feature construction might be more effective if the distribution was biased to generate more of the difficult examples.

Accordingly, we created an example generator that produced relatively more examples of illegal positions where the black king was in check by the rook. We also biased the example generator to generate many near-misses of this case - situations where the black king would be in check by the rook if the white king were not in the way. To improve the efficiency of the data generator, we did not check for repeated positions in either the training or testing set. Each example was generated independently of every other example. The same distribution bias was used for the training and testing sets.

Table 4: Comparison of Feature Construction Strategies.

Condition	Train	Accuracy (%)	Confidence
control	100	94.5 ± 1.7	NA
FF1	100	93.9 ± 2.2	($p > .5$)
FF2	100	96.0 ± 1.4	($p > .005$)
control	1000	99.43 ± 0.25	NA
FF1	1000	99.75 ± 0.29	($p > .005$)
FF2	1000	99.75 ± 0.29	($p > .005$)

We ran the experiment for each of the two *find-features* strategies, and a control condition with no feature construction. The *FFI* condition used *find-feature J*, and the *FFS* condition used *find-feature2*. Each feature construction strategy was run with 100 and 1000 training items, and 10,000 test items; the experiment was repeated 20 times. The same samples were used across conditions with the same sample size. In Table 4, the correlated two-tailed t-test was used to find significance. In the first and last groups of three rows, each feature construction strategy is compared to the control; in the middle group of two rows, *FF2* is compared with *FFI*.

4.8 Results and Observations

For the biased distribution, there were significant differences for accuracy with and without feature construction. Overall, *FF2* performed best, followed by *FFI*. Interestingly, the accuracy without feature construction is about the same as for the unbiased distribution.

FFI learned with less accuracy than the control for 100 training items, but the difference is not significant. However, it did learn with significantly better accuracy for the larger sample size. *FF2* learned with significantly better accuracy than the control for both sample sizes, and significantly better accuracy than *FFI* for the smaller sample size.

These results suggest that feature construction is dependent on the example distribution. On the other hand, it may be quite easy to bias the distribution - one can use the decision tree as a test of whether or not an example is difficult. Namely, examples that are incorrectly classified by a first-attempt decision tree might be used to construct a biased distribution for feature construction.

The difference between the two feature construction results seems to indicate that the context of the literal is important for feature construction when learning from smaller sample sizes.

5 Conclusions

STRUCT is a structural decision tree algorithm that provides good performance on the two test domains. On the kinship domain, it learns more accurate descriptions than FOIL. On the chess domain, it learns descriptions of comparable accuracy to FOIL. However, FOIL is considerably faster than STRUCT. This can be important if there are many examples. In addition, FOIL generally finds simpler definitions than STRUCT.

Our experiments demonstrated that Quinlan's heuristic of assigning a small credit to new variables does not work effectively in STRUCT for these datasets. Assigning a small debit to new variables produced more accurate decision trees. We conjecture that these results would be reversed for a separate-and-conquer strategy: Quinlan's evaluation function would consistently outperform our evaluation function because of the asymmetry of the algorithm.

The feature construction results indicate that the lack of effectiveness of feature construction on real-world domains may be due to the distribution of the examples. In order to use feature construction effectively, it may be necessary to bias the distribution. Our results showed that a straightforward extension of FRINGE for structural decision trees produced significantly less accurate descriptions for smaller sample sizes than an alternate approach that compensated for loss of context information.

Acknowledgements

Discussions with Pat Langley, Sudhakar Yerramareddy, and members of the Inductive Learning Group contributed to this paper. We would also like to thank Ross Quinlan for sending us FOIL.2.1.

References

[Bergadano and Giordana, 1988] F. Bergadano and A. Giordana. A knowledge intensive approach to concept induction. In J. Laird, editor, *Proceedings of the Fifth International Conference on Machine Learning,* pages 305-317. Morgan Kauffman, June 1988.

[Dietterich and Michalski, 1981] T.G. Diettench and R. S. Michalski. Inductive learning of structural descriptions. *Artificial Intelligence,* 16(3):257-294, 1981.

[Haussler, 1989] D. H aussler. Learning conjunctive concepts in structural domains. *Machine Learning,* 4:7-40, 1989.

[Hinton, 1986] G.E. Hinton. Learning distributed representations of concepts. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society,* Amherst, MA, 1986. Lawrence Erlbaum.

[Iba *et al,* 1988] W. Iba, J. Wogulis, and P. Langley. Trading off simplicity and coverage in incremental concept learning. In J. Laird, editor, *Proceedings of the Fifth International Machine Learning Conference,* pages 73-79. Morgan Kaufmann, June 1988.

[KodratofT and Ganascia, 1986] Y. Kodratoff and J-G. Ganascia. Improving the generalization step in learning. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning: An AI Approach,* volume 2, pages 215-244. Morgan Kaufmann, 1986.

[Manago, 1989] M. Manago. Knowledge intensive induction. In A. M. Segre, editor, *Proceedings of the Sixth International Workshop on Machine Learning,* pages 151 155. Morgan KaufTman, June 1989.

[Matheus, 1990] C. J. Matheus. Adding domain knowledge to SBL through feature construction. In *Proceedings of the Eighth National Conference on Artificial Intelligence,* pages 803-808, 1990.

[Muggleton *et al.* , 1989] S. Muggleton, M. Bain, J. Hayes-Michie, and D.Michie. An experimental comparison of human and machine learning formalisms. In A. M. Segre, editor, *Proceedings of the Sixth International Machine Learning Workshop,* pages 113-118. Morgan Kaufmann, June 1989.

[Muggleton, 1990] S. Muggleton. Inductive logic programming. In *Proceedings of the First Conference on Algorithmic Learning Theory,* Tokyo, Japan, 1990. Ohmsha.

[Pagallo and Haussler, 1990] G. Pagallo and D. Haussler. Feature discovery in empirical learning. Technical Report UCSC-CRL-90-27, University of California at Santa Cruz, Santa Cruz, CA, August 1990.

[Quinlan, 1983] J.R. Quinlan. I .earning efficient classification procedures and their application to chess endgames. In Carbonell Michalski and Mitchell, editors, *Machine Learning.* Tioga Press, Palo Alto, CA, 1983.

[Quinlan, 1990a] J.R. Quinlan. Learning logical definitions from relations. *Machine Learning,* 5:239-266, 1990.

[Quinlan, 1990b] J.R. Quinlan. I .earninc; relations: Comparison of a symbolic and a connectionist approach. Technical Report TR-346, University of Sidney, Sydney, Australia, 1990.

[Yang *et al.,* 1991] Der-Shung Yang, Larry A. Rendell, and Gunnar Blix. A scheme for feature construction and a comparison of empirical methods. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence,* Sydney, Australia, 1991.