Approximate Match of Rules Using Backpropagation Neural Networks

Boonserm Kijsirikul (boonserm@mind.cp.eng.chula.ac.th),
Sukree Sinthupinyo (g40ssp@mind.cp.eng.chula.ac.th) and
Kongsak Chongkasemwongse (g41kck@mind.cp.eng.chula.ac.th)
Department of Computer Engineering, Chulalongkorn University, Phayathai Rd., Phathumwan, Bangkok, 10330, Thailand

December 10, 2000

Abstract. This paper presents a method for approximate match of first-order rules with unseen data. The method is useful especially in case of a multi-class problem or a noisy domain where unseen data are often not covered by the rules. Our method employs the Backpropagation Neural Network for the approximation. To build the network, we propose a technique for generating features from the rules to be used as inputs to the network. Our method has been evaluated on four domains of first-order learning problems. The experimental results show improvements of our method over the use of the original rules. We also applied our method to approximate match of propositional rules converted from an unpruned decision tree. In this case, our method can be thought of as soft-pruning of the decision tree. The results on multi-class learning domains in the UCI repository of machine learning databases show that our method performs better than standard C4.5's pruned and unpruned trees.

Keywords: approximate match, feature generation, inductive logic programming, backpropagation neural networks

1. Introduction

The advantages of *inductive logic programming (ILP)* (Quinlan, 1990; Muggleton, 1991; Muggleton & De Raedt, 1994) are the expressive power of first-order logic representation and the ability of employing background knowledge. ILP systems use background knowledge provided in form of first-order logic to generalize training examples, and produce rules that fit the training examples. However, when we apply an ILP system to a real-world domain, especially the domain where there are several classes of examples or the noisy domain, the produced rules may not cover or may not exactly match with the unseen data.

Consider for example the task of learning rules for the recognition of English uppercase characters. In this task, there are 26 classes of examples, i.e. 26 different English characters, and the real-world data usually contains noise such as noise due to the quality of the scanner. To classify English characters, we may use an ILP system to learn rules for each class. With exception of some systems which learn multi-class


© 2000 *Kluwer Academic Publishers. Printed in the Netherlands.*

concepts (Baroglio & Botta, 1996; De Raedt & Laer, 1995; Martin & Vrain, 1995; Blockeel & Raedt, 1997), most ILP systems work with two classes of examples (positive and negative) and construct a set of rules for the positive class. Any example not covered by the rules is classified as negative. If we want to employ these two-class systems to learn a multi-class concept, we could do this by first constructing a set of rules for the first class with its examples as positive and the other examples as negative, then constructing the sets of rules for the other classes by the same process. The learned rules are then used to classify future data, and the rule that covers or exactly matches the data can be selected as the output. One major problem of this method is that some test data, especially noisy data, may not be covered by any rule. Thus the method is unable to determine the correct rule. A commonly used technique for solving this problem is to assign the majority class recorded from training data to the test data that is not exactly matched against any rule (Clark & Niblett, 1989; Dzeroski et al., 1996).

In this paper, we are interested in solving this problem by finding the rule that provides the best match with the data. The main contribution of the paper is to present a method for the approximate match of ILP rules by using a Backpropagation Neural Network (BNN) in case of multi-class problems or noisy domains. The basic idea is that when there is no rule covering an example, we can make use of rules which *partially* match with (partially cover) the example. Some of the partially matching rules may capture important *features (properties)*, and some may capture unimportant features of the examples. The best rule then should be the rule that matches many important features and does not necessarily match unimportant ones. The significance level of each feature is determined in terms of a weight that is trained by the BNN. Our method can deal with a related problem when a test data is covered by multiple rules.

To find the best matching rule, we also propose a novel technique for generating features from a first-order rule. The feature generation is based on the notions of *closed chains* and *open chains*. The chains define parts of rules (features) that are considered to be useful for checking properties of the examples. The features are then used by the BNN for the approximate match. We evaluate our method on four first-order datasets. The results show improvements of our method over the use of the original rules.

We also applied our method to approximate match of propositional rules converted from an unpruned decision tree. In this case, our method can be thought of as *soft-pruning* of a decision tree. Compared with standard C4.5's pruned trees on twenty datasets of multi-class learning domains in the UCI repository of machine learning databases (Merz et

al., 1997), our method performs significantly better on five domains, worse on one domain and equally well on the rest.

The paper is organized as follows. Section 2 describes the method for generating features from rules and the structure of the neural network. Section 3 and 4 describes the results on first-order and propositional learning problems, respectively. Section 5 describes related and future work. Finally the conclusion is given in Section 6.

2. Approximate Match of ILP Rules Using Backpropagation Neural Networks

Several works have shown that combining neural networks with symbolic rules produced excellent performance (Towell & Shavlik, 1994; Mahoney & Mooney, 1994; Botta et al., 1997). In this paper, a multilayer feedforward neural network is employed to select the rule that best matches with the input data. The algorithm for training the network used in our method is the backpropagation algorithm (Rumelhart et al., 1986) that is widely applied to various classification problems.

The following subsections explain the methods for generating features, building a network from features, and training the network.

2.1. Feature Generation

Our method is based on the idea that when there is no rule covering an example, we can make use of rules which *partially* cover (partially match with) the example, i.e. rules of which some literals are true for that example. The partially matching rule should not be neglected as it may capture some important properties or features of the example. The best rule should be the rule that matches many important features and does not necessarily match with unimportant ones. The significance level of each feature is determined in terms of a weight trained by the BNN which will be described later.

First, consider a first-order rule of which every literal in the body of the rule has only variables occurring in the head. For example, consider the following rule:

$$\texttt{mesh(A,11)} \leftarrow \texttt{long(A), one\_side\_loaded(A), fixed(A).}$$

Each literal checks a feature of an example. In such a rule, we will use each literal as a feature. We call this kind of feature *singleton feature*.

In a propositional rule, we can see that each feature examines a particular value of an attribute, such as the feature `outlook=sunny` in the following rule:

$$\texttt{class=play} \leftarrow \texttt{outlook=sunny, humidity} \leq 75.$$

However, it is more difficult to determine what should be used as features when we consider first-order rules with new variables. A literal with new variables itself may not check for a specific property of the example, i.e. the literal alone may be meaningless without the presence of other literals which make use of the newly introduced variables. Most literals introducing new variables are for passing the introduced variables to other literals that may check a property or introduce other new variables again. Usually a newly introduced variable should end at a literal that checks for a property. The connection of the variables via the sequence of literals thus examines a feature of the example. Below we give an algorithm to select a sequence of literals to be used as a feature. Our method of feature generation is based on the notion of *closed* and *open chains*.

Definition 1 *(Closed chain).* A sequence of some literals in the body of a rule is said to be a *closed chain* if every new variable not occurring in the head of the rule appears at least in two literals of the sequence and occurs at least once in a literal with variable(s) of the head or with variable(s) in one of the preceding literals.

Intuitively speaking, a new variable not occurring in the head of a rule is in a closed chain if after it is introduced by a literal it must be consumed by another literal. The closed chain does not allow a variable which occurs alone in two or more literals without being linked to existing variables. However, in some cases, some variables may not be in a closed chain.

Definition 2 *(Open chain).* A sequence of some literals in the body of a rule is said to be an *open chain* if there exists a new variable not occurring in the head of the rule which appears only once in a literal with variable(s) of the head or with variable(s) in one of the preceding literals.

Some examples of closed and open chains are shown below.

Example 1 *(Closed chain).* For the rule:

```
p(A,B)← q1(A), q2(A,C), q3(C), q4(C,D), q5(D), q6(A,E,F),
        q7(E,G), q8(E,H), q9(E), q10(F,I), q11(I,B).
```

some closed chains are:

(i) `q2(A,C), q3(C)`
(ii) `q2(A,C), q4(C,D), q5(D)`
(iii) `q2(A,C), q3(C), q4(C,D), q5(D)`
(iv) `q6(A,E,F), q9(E), q10(F,I), q11(I,B)` □

Example 2*(Open chain).* For the rule in Example 1, some open chains are:

(i) `q6(A,E,F), q7(E,G)`
(ii) `q6(A,E,F), q8(E,H)`
(iii) `q6(A,E,F), q7(E,G), q8(E,H)`
(iv) `q6(A,E,F), q7(E,G), q9(E)`
(v) `q6(A,E,F), q8(E,H), q9(E)`
(vi) `q6(A,E,F), q7(E,G), q8(E,H), q9(E)` □

The definition of the open chain does not allow the variable that occurs only once in the chain but is not linked to other variables occurring in the head or in the preceding literals. For example, the open chains, for the rule in Example 1, do not include the sequences "`q3(C)`", "`q1(A), q3(C)`", "`q5(D)`", "`q1(A), q5(D)`", etc.

We now describe our method for generating features of a rule. The method is best understood by viewing a rule as a dependency graph. The root node of the graph is a set of variables occurring in the head of the rule. Each of the other nodes represents a set of new variables introduced by a literal, and an edge to the node represents the literal. The whole graph shows the dependency of variables. Figure 1 shows an example of dependency graph of the rule in Example 1.

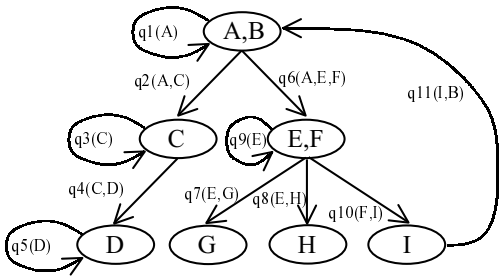

Figure 1. The dependency graph for the rule "`p(A,B)←q1(A), q2(A,C), q3(C), q4(C,D), q5(D), q6(A,E,F), q7(E,G), q8(E,H), q9(E), q10(F,I), q11(I,B).`''

Using the definitions of closed and open chains and viewing a rule as a dependency graph, we can now describe our algorithm for generating features as shown in Table I.

Table I. The algorithm for feature generation.

1.	Find every edge beginning and ending at the root node and use it as a feature. Remove this kind of edges from the graph and do not consider the edges in the following steps. This type of feature is a *singleton feature* which introduces no new variable.
2.	Find all possible closed chains starting from the root node, and use the sequence of literals along each of the chains as a feature. This type of feature is a *closed chain feature*.
3.	For every leaf node that has no edge to others, find all possible paths that start from the root to that node. Use the sequence of literals along each of the paths as a feature. This type of feature is an *open chain feature*.
4.	Find every possible combination of the open chain features generated in Step 3 that have new variables (not occurring in the head) in common. If the combination is different from the existing open chain features, then add it to the feature set.

The algorithm in Table I generates all closed chains that include variables at the root node of the graph. However, it does not generate all possible open chains; it does not generate open chains which are sub-chains of a closed chain feature. This is because we consider that usually a newly introduced variable should be consumed by another literal that checks for a specific property of the example. Therefore, we first generate all closed chain features if they exist; open chains which are sub-chains of a closed chain feature will not be generated. However, for some literal which introduces new variables, we may be unable to find a closed chain feature for the literal. In such a case, we generate an open chain feature that includes the literal. An example of feature generation is shown in Example 3.

Example 3 *(feature generation).* For the rule in Example 1, all possible features generated by our algorithm are:

Step 1. in Table I

(i) `q1(A)`

Step 2. in Table I

(ii) `q2(A,C), q3(C)`

(iii) `q2(A,C), q4(C,D), q5(D)`

(iv) `q2(A,C), q3(C), q4(C,D), q5(D)`

(v) `q6(A,E,F), q9(E), q10(F,I), q11(I,B)`

Step 3. in Table I

(vi) `q6(A,E,F), q7(E,G)`

(vii) `q6(A,E,F), q8(E,H)`

(viii) `q6(A,E,F), q9(E), q7(E,G)`

(ix) `q6(A,E,F), q9(E), q8(E,H)`

Step 4. in Table I

(x) `q6(A,E,F), q7(E,G), q8(E,H)`

(xi) `q6(A,E,F), q7(E,G), q8(E,H), q9(E)` □

Our method of feature generation can also be thought of as the transformation of a rule into an equivalent rule with duplicated literals that preserves the meaning of the original rule. The method reformulates the original rule into a number of parts, by using singleton, closed and open chain features, each of which is for examining a particular property of the example. For instance, using features in Example 3, the rule in Example 1 will be transformed into:

```
p(A,B)← q1(A),
        q2(A,C), q3(C),
        q2(A,C), q4(C,D), q5(D),
        q2(A,C), q3(C), q4(C,D), q5(D),
        q6(A,E,F), q9(E), q10(F,I), q11(I,B),
        q6(A,E,F), q7(E,G),
        q6(A,E,F), q8(E,H),
        q6(A,E,F), q9(E), q7(E,G),
        q6(A,E,F), q9(E), q8(E,H),
        q6(A,E,F), q7(E,G), q8(E,H),
        q6(A,E,F), q7(E,G), q8(E,H), q9(E).
```

The obtained rule is then used to construct the structure of a neural network for enabling the approximate match, as described in the next subsection.

2.2. Building a Neural Network from Features

As some ILP systems are able to specialize variables to constants and unify two variables, the head of a rule may contain constants or the same variables occurring in more than two arguments. The following rule is such an example:

```
illegal(WKf,3,WRf,WRr,BKf,WRr)←lt(WKf,3).
```

This rule contains a constant "3" at the second argument and a variable "`WRr`" at the fourth and sixth arguments. Before we map rules to a neural network, we first preprocess such a rule by using the unification (=) so that the head of the rule does not contain constants and the same variables. This is done by replacing constants and the same variables in the head with all different variables and adding the unifications into the body which unify the constants with the head variables and unify pairs of the head variables.

Using this preprocessing, the above rule will be converted to:

`illegal(WKf,WKr,WRf,WRr,BKf,BKr)` ← `WKr=3, WRr=BKr, lt(WKf,3).`

This preprocessing is needed because the specialization or unification can be viewed as an operator that checks for a property of the argument and thus should be considered as a feature. However, we do not replace the constant in the head if the constant defines the class. For example, the constant "1" in the rule "`mesh(A,1)` ← `not_important(A), not_loaded(A).`" indicates the class, and is not replaced by a variable.

Given a set of rules obtained from the preprocessing technique, we then generate the singleton, closed chain and open chain features for each rule by using the algorithm in Table I. The features of a rule are used as input units that are linked to one hidden unit which represents the rule. Therefore, the number of hidden units in the network is the same as the number of rules. Each class is represented by one output unit of the network. In two-class problems, there are two output units, one for the positive and the other for the negative class. In multi-class problems, the number of output units is equal to the number of classes. The links from hidden units to output units are fully connected. Note that all hidden and output units include bias weights. All weights of all links and bias weights are trained by the backpropagation algorithm.

Example 4 *(building a neural network from features).* Consider the following rule set $\{C1, C2, C3, C4\}$ in the "finite element mesh design" problem (see Section 3.2).

C_1 : `mesh(A,1)` ← `not_important(A), not_loaded(A).`
C_2 : `mesh(A,2)` ← `short(A), opposite_l(B,A).`
C_3 : `mesh(A,2)` ← `usual(A), neighbour_yz_r(A,B), cont_loaded(B).`
C_4 : `mesh(A,3)` ← `short(A), neighbour_zx_r(A,B), opposite_r(A,C), short(C).`

The features generated for each rule are as follows, where F_iC_j is the i^{th} feature of the rule C_j.

F_1C_1: `not_important(A)`

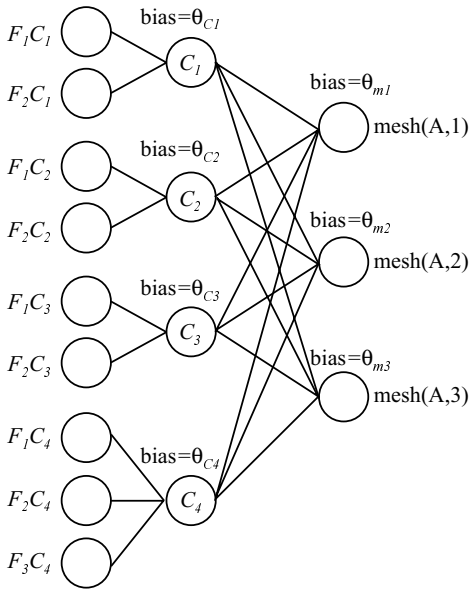

Figure 2. The structure of the neural network for the rule set $\{C_1, C_2, C_3, C_4\}$ in Example 4.

F_2C_1: `not_loaded(A)`

F_1C_2: `short(A)`
F_2C_2: `opposite_l(B,A)`

F_1C_3: `usual(A)`
F_2C_3: `neighbour_yz_r(A,B), cont_loaded(B)`

F_1C_4: `short(A)`
F_2C_4: `opposite_r(A,C), short(C)`
F_3C_4: `neighbour_zx_r(A,B)`

In this example, most of the features are singleton features. F_2C_3 and F_2C_4 are closed chain features, and F_2C_2 and F_3C_4 are open chain features. A singleton feature examines a property of an example by using a single literal, such as F_2C_1:"`not_loaded(A)`" checking for the property that the edge "`A`" has no loading. A closed chain feature is useful for checking the property that is defined by two or more literals. F_2C_4 examines the property that an edge "`C`" which is opposite to "`A`" must be a short edge. "`Short(C)`" in F_2C_4 will be meaningless, if it is solely considered as a feature. The open chain feature F_3C_4 cannot be neglected, and it examines the property that there is some edge "`B`" which is a neighbour of "`A`".

Assume that there are only three classes, i.e. `mesh(A,1)`, `mesh(A,2)` and `mesh(A,3)`. Figure 2 shows the structure of the network for the

above rules. The hidden unit C_1 representing the rule C_1 is linked to two input units each of which is a feature of C_1. Similarly, the hidden units C_2, C_3 and C_4 are linked to their features. In this example, there are three output units representing the classes. □

When new variables are considered, there can be many variable bindings for a rule that make different truth values for literals containing such variables. In such a case, there can be a number of possible ways to select bindings. For example, one would use the binding that gives the minimum number of features whose truth values are true, or use bindings that assign *true* to many *important* features and few unimportant features. The use of the minimum number of *true-features* (features whose truth values are true) does not seem to be a good choice. The binding which assigns *true* to many important features and few unimportant features is a better choice. However, before the network is completely trained, it is not known in advance which features are important. Even after the network is completely trained, it is still difficult and costly to find and separate important features from unimportant ones. Therefore, in our implementation, we use a reasonable and simple method that selects the binding which gives the maximum number of features whose truth values are true. The truth value of a feature is true if the truth values of all literals of the feature are true, otherwise the truth value of the feature is false. In case there is more than one binding that gives the same value of the maximum number of true-features, we just randomly choose only one.

2.3. Training the Network

The weights of the network are randomly initialized, and the final weights are obtained by the standard backpropagation algorithm (Rumelhart et al., 1986). In our experiment, all units in the network use the sigmoid function.

The training examples of the network are basically the same as those used to create a rule set, except for the following points.

- For multi-class problems, only examples that are covered by rules are used. As the learned rules are not representative for uncovered examples, they are not used to train the network.

- For two-class problems, all negative examples are used, and only positive examples covered by rules are used as the learned rules are representative only for the covered positive examples.[1]

[1] We also ran experiments by using *all* positive and all negative examples, and found that the results are almost the same.

Each training example is evaluated with every rule and the truth values of features are determined. We use a Prolog interpreter to evaluate the truth values of features, and thus our method works with Horn clauses including recursive rules. The features whose truth values are true are set to 1, whereas the features whose truth values are false are set to -1 for input units. The network is repetitively trained by using training examples until it converges or the number of training iterations exceeds the predefined threshold. After having been trained, the network can be used to classify unseen data. The unseen data is evaluated with features of each rule as in the training process. The truth values of features are then fed into the network, and the output with highest value will be taken as the prediction.

3. Results on First Order Domains

We implemented a learning system, BANNAR (Backpropagation Artificial Neural Networks for Approximating Rules). BANNAR receives a rule set, training examples and background knowledge as the inputs. The system uses the rule set to construct a feature set for building a neural network, and trains the network by using background knowledge to determine the truth values of features for each example.

In the following experiments, we selected PROGOL (Muggleton, 1995) or GOLEM (Muggleton & Feng, 1990) for learning rules. Normally we used rules produced by PROGOL as the input to BANNAR. However in our experiments on the *finite element mesh design* and the *King-Rook-King chess endgame* datasets described below, PROGOL failed to produce a rule set within a reasonable time. In those experiments, we employed GOLEM developed by the same research group of PROGOL. We then compared the results obtained by BANNAR with those of the original rule set. To show the quality of the rule set used in our method, we also included the results obtained by the other three learning systems, i.e. TILDE, 1BC and LINUS. TILDE is a multi-class learning system that extends C4.5 to a first-order decision tree learner (Blockeel & Raedt, 1997). 1BC is a first-order probabilistic learning system using the naive Bayes algorithm (Flach & Lachiche, 1999). LINUS is a first-order learner based on the transformation approach (Lavrač & Džeroski, 1994).

In the following subsections, we briefly describe the learning systems and the datasets, and present the experimental results. Next, to understand the characteristic of BANNAR, we run additional sets of experiments and report the results in Section 3.4, 3.5 and 3.6.

3.1. Learning Systems

PROGOL

PROGOL is a state-of-the-art ILP system described in (Muggleton, 1995). Having positive examples, negative examples, background knowledge and *mode declarations* as inputs, PROGOL constructs a most specific rule for a random seed example. The mode declarations specify, for each argument of each predicate, the type of the argument and whether it should be a constant, a variable bound before the predicate is called, or a variable bound by the predicate. Given a most specific rule, PROGOL performs a complete search of the hypothesis space bounded below by the most specific rule, using A*-like search (Nilsson, 1980). Because of its complete search, PROGOL usually requires more time and memory than the other systems used in our experiments.

Although PROGOL is able to learn from only positive examples, the system usually works well with two-class (positive and negative) examples. Therefore, in our experiment on the *Thai character recognition* dataset (see below) which contains 77 classes of examples, a set of rules is induced for each class. Positive examples of one class are treated as negative examples of the other classes.

GOLEM

GOLEM (Muggleton & Feng, 1990) is based on the concept of relative least general generalization (rlgg) (Plotkin, 1970) with the *ij-determinate* constraint that is used to restrict the class of programs to be efficiently learned. The inputs to GOLEM are positive examples, negative examples and background knowledge. To learn a rule, GOLEM randomly selects *several* pairs of positive examples and computes the rlggs of each pair. The resulting rlgg, which is represented as a first-order rule, with the greatest coverage of positive examples is selected, and that rule is further generalized by computing the rlggs of the rule with new randomly chosen positive examples. The generalization process stops when the coverage of the best rule does not increase.

The user can specify the number of pairs to be considered for constructing rlggs (the default setting is 8). If the number of pairs is large, GOLEM will have a better chance of finding good rules. In the experiments in Section 3.3, we use the default setting. GOLEM is a two-class learning system, and needs negative examples to learn rules. In the experiment on the *finite element mesh design* dataset (see below), negative examples are generated by using the closed-world assumption.

TILDE

Unlike PROGOL and GOLEM that learn rules, TILDE learns logical decision trees (Blockeel & Raedt, 1997). TILDE extends the attribute-value learner C4.5 to a first-order learning system. The learning algorithm used in TILDE is similar to C4.5. TILDE starts with the empty tree, and then generates all possible test nodes and computes the heuristic values of these test nodes. However, rather than using attribute-value tests in the nodes, TILDE employs first-order logical queries. Among the possible test nodes, it will select the one that scores best on the heuristic and place that in the current node. It will then use the partial tree to classify all examples (in the current node) to its sub-nodes. All examples passing the test will be propagated to the left, all the other ones to the right. The procedure will then recursively analyze the left and right sub-trees. When a node only contains examples of a single class (or when heuristics indicate it is uninteresting to split a node), it will be turned into a leaf. Note that TILDE is a multi-class learning system.

1BC

1BC is a first-order probabilistic learning system based on the naive Bayes algorithm. The naive Bayes algorithm works by collecting statistical information in training data, and uses this information to classify unseen data. The statistical information collected are (1) the probability of encountering each class, (2) the probability of seeing each feature given a particular class. To predict the class of an unseen data, the algorithm then finds the class that maximizes the product of the above two kinds of probabilities. See (Flach & Lachiche, 1999) for details. In propositional learning, a feature is a test for a specific value of an attribute, such as `attribute1=value1`. However, in first-order learning, like the problem faced in our method, 1BC has to determine what will be used as features. 1BC employs *structural predicates* and *properties* provided by users for generating *atomic features*. As the number of atomic features can be large, 1BC constrains this number by allowing the user to limit the numbers of variables and literals for each atomic feature. Atomic features containing more variables or literals than the user-specified value will not be considered. In our experiments, we asked 1BC to generate atomic features containing up to three literals and three variables. However, in the dataset of the finite element mesh design, due to memory of our computer system, we had to limit these numbers to two.

LINUS

LINUS is a first-order learning system based on the *transformation approach* (Lavrač & Džeroski, 1994). The system works by first transforming a first-order problem to an attribute-value one, and then learning with an attribute-value learner. LINUS can transform the learned description back to first-order rules. The system first transforms a first-order example with background knowledge into a set of truth-value tuples described by *features (literals)*. Depending on background knowledge, the number of generated features may be very large. To reduce the number of generated features, Lavrač et al. (Lavrač et al., 1999) propose a technique for selecting only *relevant* features and eliminating irrelevant ones.

The attribute-value learner employed by LINUS in our experiments is C4.5 because of its availability and its multi-class learning ability. The accuracy of LINUS in our experiments is evaluated by C4.5's trees.

3.2. Datasets

Thai Character Recognition

The dataset consists of 77 classes of examples, i.e. 77 different Thai characters[2]. The goal of this task is to learn rules for predicting the class of unseen data. In the training set, each character has 14 examples constructed from 14 sample images. The total number of training examples is 1,078. The noise was added into the original images, and the test data were constructed. The test set contains 2,143 test examples. This is a usual experimental setting in the task of the character recognition as the learned rules or neural networks usually encounter unseen images containing noise when they are used by a character recognition software.

Each example is of the form `char(A,B,C,D,E,F)`. The information contained in `A,B,C,D,E,F` are *image features*[3] extracted by a preprocessing algorithm. These image features describe various properties of a character image such as the ratio of the width and the height of the character, the structure of lines and circles that form the character, the list of zones in the images that contain junctions of lines, etc. The background knowledge contains 55 predicates. See (Kijsirikul & Sinthupinyo, 1999) for more details.

[2] The dataset will be made available at http://mind.cp.eng.chula.ac.th.

[3] These are features describing the structure of the character images, such as the ratio of the width and the height of the character image. These features should not be confused with the feature generation described in Section 2.1.

Finite Element Mesh Design

The dataset for the finite element mesh design (Dolsak & Muggleton, 1992) consists of 5 structures and has 13 classes (13 possible number of partitions for an edge in a structure). Each example is of the form `mesh(Edge,Number)` where `Number` indicates the number of partitions. The total number of examples is 278. The goal of finite element mesh design is to learn general rules describing how many elements should be used to model each edge of a structure. The background knowledge consists of relations describing the properties of an edge (e.g. `short`, `not_loaded`), boundary conditions (e.g. `free`), loadings (e.g. `not_loaded`), and the relations describing the structure of the object (e.g. `neighbour`). See (Dolsak et al., 1994; Dolsak & Muggleton, 1992) for more details.

Mutagenesis

The dataset for the mutagenesis domain consists of 188 molecules, of which 125 are active and 63 are inactive. The goal of this problem is to predict the mutagenicity of the molecules, whether a molecule is active or inactive in terms of mutagenicity. This problem is a two-class learning problem. A molecule is described by listing its atoms `atom(AtomID,Element,Type,Charge)` and the bonds `bond(Atom1,Atom2,BondType)` between atoms. The background knowledge used in our experiment is the set $S2$ described in (Srinivasan et al., 1996) that contains the definition of atom, bond, methyl groups, nitro groups, aromatic rings, hetero-aromatic rings, connected rings, ring length, and the three distinct topological ways to connect three benzene rings. See (Srinivasan et al., 1996) for more details.

King-Rook-King Chess Endgame

The last dataset used in our experiment is the King-Rook-King chess endgame (KRK) dataset provided by the Machine Learning group at the University of York[4]. The goal is to learn the concept of an illegal white-to-move position with only white king, white rook and black king on the board (Muggleton et al., 1989). The number of examples in the dataset is 10,000; 3,240 representing illegal KRK endgame positions (positive), the rest representing legal endgame positions (negative). Each example is of the form `illegal(WKf,WKr,WRf,WRr,BKf,BKr)`, where `(WKf,WKr)`, `(WRf,WRr)` and `(BKf,BKr)` are the positions (file,rank) of White King, White Rook and Black King, respectively. Each of these

[4] http://www-users.cs.york.ac.uk/~stephen/chess.html

variables takes values from 0 to 7. Background knowledge contains two relations for comparing rank and file; `adj(X,Y)` and `lt(X,Y)` indicating that rank/file `X` is adjacent to rank/file `Y` and rank/file `X` is less than rank/file `Y`, respectively.

Note that only in this dataset, for 1BC we used the background relations described in (Flach & Lachiche, 1999), such as `board2whiteking, board2blackking, board2whiterook, fileeq, rankeq, pos2rank`, etc. For the other datasets described above, all background relations given to all learning systems are the same.

3.3. Experimental Results on First-Order Datasets

We used three-fold cross-validation[5] and averaged the results in all experiments except for the experiment on the Thai character recognition dataset where training and test data were given. For the accuracies of PROGOL or GOLEM, we assigned the majority and the negative class to examples uncovered by the rules in the case of multi-class and two-class problems, respectively. The experimental results reporting the accuracies of all systems are shown in Table III [6]. Table II reports the training times required for training BANNAR and PROGOL (GOLEM).

Table II. The training times in seconds of BANNAR and PROGOL (GOLEM), the number of features generated by BANNAR and the number of rules generated by PROGOL (GOLEM). All systems were run on 400Mhz Pentium II. The times, the number of features and the number of rules are the average values for three-fold data, except for the TCR data set.

Data Set	BANNAR		PROGOL or GOLEM	
	# Features	Time (sec.)	# Rules	Time (sec.)
TCR	467.0	39.38	77.0	1344112.92
FEM	115.0	32.98	38.0	3607.88
MUTA	18.3	12.67	6.3	6014.30
KRK	32.3	925.03	14.0	36.00

Though our method required an additional time to train BANNAR, it was shown in Table II that, except for the KRK dataset, the training time of BANNAR was much less than that of PROGOL or GOLEM. PROGOL or GOLEM required quite a long time, especially on the

[5] As training ILP systems requires quite a long time, we used only three folds in the cross-validation experiments.

[6] The results were obtained by using the default settings of all systems.

Table III. The percent accuracies of BANNAR and the other systems on first-order datasets; TCR – Thai Character Recognition, FEM – Finite Element Mesh Design, MUTA – Mutagenesis, KRK – King-Rook-King Chess Endgame. Superscripts denote confidence levels for the difference in accuracy between BANNAR and the corresponding system, using a one-tailed paired t test: * is 90.0%, ** is 99.0%, *** is 99.5%; no superscripts denote confidence levels below 90%. 3CV denotes the experiment that uses three-fold cross-validation. The accuracies of PROGOL (GOLEM) are calculated by assigning the majority and negative class to uncovered examples in the case of multi-class and two-class problems, respectively.

Data Set	#Train	#Test	#Classes	BANNAR	PROGOL or GOLEM	TILDE	1BC	LINUS
TCR	1,076	2,143	77	94.40	72.00***	88.57***	77.23***	66.54***
FEM	278	3CV	13	64.45	57.80**	58.02**	46.73**	60.45*
MUTA	188	3CV	2	83.58	82.01	68.94*	77.72	74.41*
KRK	10,000	3CV	2	99.90	99.83	69.67***	87.11***	99.36***

TCR dataset. In the TCR dataset, as there are 77 classes of examples, PROGOL used a significant time to learn rules for 77 classes. The training time of BANNAR is proportional to the number of training examples and the number of rules, because it needs to match features of all rules with the examples. Therefore, in the case of the KRK dataset in which there are 10,000 examples, BANNAR used a longer time to run than in the case of the other datasets.

The results in Table III show that the performance of PROGOL or GOLEM was comparable to that of TILDE; PROGOL or GOLEM performed better than TILDE in two-class problems, whereas TILDE did better in multi-class problems. In the datasets tested in our experiments, 1BC did not perform well, compared to PROGOL or GOLEM. LINUS performed better than PROGOL or GOLEM only on the FEM dataset and worse on the other datasets. These results show the high accuracies of rules produced by PROGOL or GOLEM. Nevertheless, as shown in the table, BANNAR was still able to improve the accuracies of the rules, especially in the multi-class problems. Compared with the other learning systems, BANNAR performed best on all datasets. Moreover, BANNAR significantly outperformed PROGOL or GOLEM, TILDE, 1BC and LINUS on 2, 4, 3 and 4 datasets, respectively. Note that in the MUTA dataset, there are cases that multiple rules fire but there is no difficulty for BANNAR as it predicts the class which best matches the examples.

We further investigate these improvements. We want to see how well BANNAR classifies examples when they are not covered by the rules. Table IV summarizes the results.

Table IV. Improvements of BANNAR over the original rules, reported according to covered and uncovered examples. The columns "Covered" and "Uncovered" denote the numbers of examples covered and uncovered by the rules. Each cell denotes the number of examples correctly classified/the number of examples for that portion.

Data Set	# Test	BANNAR		PROGOL or GOLEM (Majority or Negative Class)	
		Covered	Uncovered	Covered	Uncovered
TCR	2,143	1570/1611	453/532	1540/1611	3/532
FEM	278	145/200	34/78	146/200	14/78
MUTA	188	102/109	55/79	100/109	54/79
KRK	10,000	3352/3356	6638/6644	3350/3356	6633/6644

The results in the table show the ratio between the number of examples correctly classified and the number of examples for each portion. For example, 453/532 in the row TCR indicates that 532 examples were not covered by the rules, and 453 of them were correctly classified by BANNAR. 3/532 in the same row shows that 3 of 532 examples were correctly classified by PROGOL as we used the majority class for predicting unseen data (or the negative class for two-class problems). This means that BANNAR correctly classified 450 more examples in the case of uncovered examples. Similarly, 1570 of 1611 examples were correctly classified by BANNAR, whereas 1540 were correctly classified by PROGOL; although the number of examples covered by PROGOL was 1611, 1540 out of them were correct. A similar improvement can be seen on the FEM dataset. Slight improvements were obtained on the MUTA and the KRK datasets which are two-class problems. These results show that BANNAR significantly improved the accuracy on data which were not covered by the rules, especially in the multi-class problems.

In the next subsection, we explore if our method will help in two-class problems with the presence of noise.

3.4. Experiments on KRK Noisy Datasets

To study the effect of noise on two-class learning problems, we selected the KRK dataset. In the following experiments, three-fold cross-validation was used. The dataset was partitioned into three disjoint subsets. Each subset was used as a test set once, and the remaining

subsets were used as the training set. Given training and test sets, 5%, 10% and 15% class noise was randomly added into the training set, and no noise was added into the test set. In our case, adding x% of noise means that the class value was replaced with the wrong value in x out of 100 data. For example, 5% of noise means that 5% of data were randomly selected and the class value of each data was replaced by the opposite value (from positive to negative, and vice versa). The average results of GOLEM[7] and BANNAR on noisy data are shown in Table V. In the table, we also included the result on noise-free data for comparison.

To see how well BANNAR handles noisy data, we also ran FOSSIL on the same dataset, and reported the results together in Table V. FOSSIL is a first-order learning algorithm that has the ability to handle noisy data by using the *correlation* heuristic to prevent learning of over-specific rules. The system has been shown to be robust against noise (Fürnkranz, 1994).

Table V. The percent accuracies (Acc.) of GOLEM, BANNAR and FOSSIL with 5%, 10% and 15% noise added. The dataset contains 10,000 examples. The experiment was run using three-fold cross-validation. The number of rules obtained by GOLEM is the average value for three-fold data.

Noise Level	GOLEM		BANNAR	FOSSIL
	#Rule	Acc.	Acc.	Acc.
0%	14.00	99.83	99.90	99.02
5%	49.67	92.27	98.09	97.93
10%	134.00	87.32	98.12	96.08
15%	150.00	82.51	94.33	94.79

As shown in the table, when noise was added, GOLEM produced a large number of over-specific rules that were needed to cover positive training data. These rules fitted only the training data well, but they resulted in wrong classification for unseen data as shown by the decrease of the accuracy with increasing noise. The results of BANNAR show that the accuracy decreased much slower than that of GOLEM, and BANNAR significantly improved the accuracy of GOLEM when noise was added (the differences are statistically significant at the confidence levels of 99.5% for 5% or 10% noise, and 97.5% for 15% noise). These results show BANNAR's robustness against noise which is useful to increase the accuracy of rules produced by a learning system having

[7] In the experiments below, the number of pairs of examples to be considered for constructing rlggs is set to 50.

no ability to handle noise, such as GOLEM. Comparing the results of BANNAR with FOSSIL which has ability to handle noise, we see that the accuracies of BANNAR were comparable to those of FOSSIL. The accuracies of BANNAR were higher than those of FOSSIL in the case of 0%, 5% and 10% noise, and lower in the case of 15% noise.

The mechanism of BANNAR to prevent overfitting noisy data is different from that of FOSSIL. Whereas FOSSIL uses the heuristic to prevent learning over-specific rules, BANNAR employs neural networks to give less attention to unimportant features. Even using over-specific rules to construct a neural network, BANNAR was able to produce the network which comparatively did not overfit the data as shown by the slower decrease of its accuracy. The following example shows that BANNAR gave appropriate weights to features, i.e. higher weights to important features and lower weights to unimportant ones. In our experiment, one of rules obtained when 5% noise was added is:

```
illegal(WKf,WKr,WRf,WRr,BKf,BKr) ←
              WRr=BKr, lt(WKr,WRr), lt(WKf,WRf).
```

The rule contains three features, and states that the position is illegal if (1) the ranks of the white rook and black king are the same, and (2) the rank of the white king is less than the rank of the white rook (thus the white king is not blocking the check), and (3) the file of the white king is less than (below) that of the white rook. Clearly, the feature (3) is not necessary if the features (1) and (2) are satisfied. This rule is an over-specific rule and it is likely that the rule overfits noisy data. The literal `lt(WKf,WRf)` should not be added to this rule, i.e. the rule will correctly classify more data if the literal is not included in the rule. When we employed BANNAR, the system found the appropriate weight for each feature of the rule. In this case, each literal was selected as a feature. The unnecessary feature, i.e. `lt(WKf,WRf)`, was given a lower weight by BANNAR. Figure 3 shows the weights of literals of the rule. As shown in the figure, the weight of unuseful literal `lt(WKf,WRf)` is -2.889 and is dominated by the sum of weights of the others which is $7.214 + 4.151 = 11.365$. Therefore, if the first two features are satisfied, the hidden unit $R1$ representing this rule will give the positive output and makes a high chance of predicting the positive class.

The ability to give appropriate weights to features is the advantage of BANNAR, because the unimportant features will receive less attention in classifying unseen data. In this case, although the original rule is over-specific, the obtained part of the network is very useful to classify unseen data.

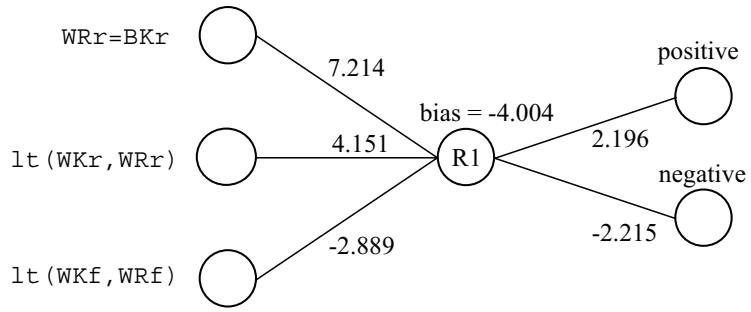

Figure 3. A portion of the network for the rule `illegal(WKf,WKr,WRf,WRr,BKf,BKr)` ← `WRr=BKr, lt(WKr,WRr), lt(WKf,WRf).`

3.5. A Comparison of BANNAR's and LINUS's Features

This subsection describes experiments which compare features generated by BANNAR and those generated by LINUS. For comparison, these two sets of features were employed by the same attribute-value learners, i.e. C4.5 and a neural network. In BANNAR, as described earlier, the features were generated from rules obtained by PROGOL (GOLEM), and the truth values of the features for examples were evaluated to be used as inputs to C4.5 or a neural network. In LINUS, features were directly generated from background knowledge, and the truth values of features for examples were evaluated to be used as inputs to the same learning algorithm. In the case of C4.5, each feature generated by BANNAR or LINUS was considered as an *attribute*. In the case of a neural network, each feature was used as an input unit of the network. A neural network constructed from LINUS's features consists of three layers, i.e. input, hidden and output layers. The number of input units and the number of output units are equal to the number of features, and the number of classes, respectively. The number of hidden units was determined to be the same as the number of input units. The links from input units to hidden units, and from hidden units to output units are fully connected. The experimental results are shown in Table VI.

The results show that compared to LINUS's features, BANNAR's features gave higher accuracy, both in the case of a neural network and C4.5. This shows the effectiveness of features generated by BANNAR. Note that BANNAR generates features from rules obtained by PROGOL (GOLEM), and thus the good quality of BANNAR's features is due in part to the high accuracy of rules produced by PROGOL (GOLEM).

Table VI. The comparison of features generated by BANNAR and LINUS. The number of features or the percent accuracy (Acc.) is the average value for three-fold data, except for the TCR data set.

Data Set	# Features		Acc. of Neural Networks		Acc. of C4.5	
	BANNAR	LINUS	BANNAR's Features	LINUS's Features	BANNAR's Features	LINUS's Features
TCR	467.0	36.0	94.40	67.10	89.22	66.54
FEM	115.0	209.0	64.45	57.97	60.46	60.45
MUTA	18.3	16.0	83.58	78.53	81.44	74.41
KRK	32.3	72.0	99.90	99.57	99.82	99.36

3.6. Empirical Study on the Improved Accuracy of BANNAR

The results on previous sections show the usefulness of our method for increasing the classification accuracy of rules for unseen data, especially data in multi-class problems and noisy domains. The good performance of BANNAR is certainly due to the high accuracy of rules produced by PROGOL or GOLEM, and more improvement comes from two main elements: (1) our feature generation that enables partial match between the examples and the useful parts of rules which examine some properties of the examples, and (2) backpropagation neural networks that make use of features to enable more flexible match of rules with the examples. Though it is difficult to separately evaluate the contribution of each of these two elements to the increased accuracy of BANNAR, in this section we try to give some empirical evaluation. We designed a set of experiments that were aimed at explaining the contribution of each element to the improved accuracy of BANNAR.

The experiments were designed to evaluate that (1) how much features alone increase the accuracy, and (2) how much neural networks further increase the accuracy.

To evaluate the effect of features alone, we employ a simple technique that makes use of features to match with examples uncovered by the rules. When there are no rules which perfectly match an unseen data, we want to select the rule that provides the best match with the data. Therefore, our technique is to find the rule that contains a lot of features matching with the data. To find the best matching rule, we define $matchRatio_{r,e}$ of a rule r for an example e as follows:

$$matchRatio_{r,e} = \frac{trueFeature(r,e)}{no.\ of\ all\ features}$$

where $trueFeature(r, e)$ is the number of features of r whose truth values are true for e. The value of $matchRatio$ will be high if the rule contains a lot of features matching with the example. In the case of multi-class problems, the best matching rule for the example e will be the rule that gives the highest $matchRatio$, and the corresponding class of the rule will be selected as the prediction.

However, in two-class problems, we have no rule for the negative class. Therefore, we use a simple method that will classify an example e as positive by using the following criterion.

$$\exists r \quad matchRatio_{r,e} > \theta_r$$

where

$$\theta_r = \max_{n \in NE} \; matchRatio_{r,n}$$

and NE is the set of all negative training examples.

θ_r is the threshold of the rule r and is defined to be the maximum value of $matchRatio$'s among all negative training examples. The value θ_r ensures that all *negative training* examples will be classified as negative. Therefore, for an example e, if there exists $matchRatio_{r,e}$ greater than θ_r, the example will be classified as positive; otherwise it will be classified as negative.

Though it did not occur in the following experiments, in case of noisy training data, there may be some negative example that is covered by a rule; the number of true-features for that example will be equal to the number of all features. For such a rule, the maximum value of $matchRatio$'s among all negative examples will be 1. In this case, we use the weak criterion that will classify an example as positive only if $matchRatio$ of that rule for the example is equal to 1 (not greater than 1).

Using this technique, in the case of multi-class problems, we evaluate $matchRatio$'s of all rules for an example to be classified, and predict as the output the class with the corresponding highest $matchRatio$. In two-class problems, we first calculate θ_r's of all rules by using the set of all negative training examples, then for an example e we evaluate $matchRatio_{r,e}$'s of all rules, and classify the example as positive if there is some $matchRatio_{r,e}$ that is greater than θ_r (or equal to 1). The results of this technique denoted as "FEATURE ALONE" are shown in Table VII. For comparison, we included the results of the original rules and the results of BANNAR in the table.

As shown in the table, the method of "FEATURE ALONE" classified more correct uncovered-examples than PROGOL or GOLEM did, especially in the multi-class problems (TCR and FEM). For example, the number of correctly classified uncovered-examples increased from

Table VII. The results of using features alone and those of PROGOL (GOLEM) and BANNAR on the first-order datasets, reported according to covered and uncovered examples. Each cell denotes the number of examples correctly classified/the number of examples for that portion.

Data Set	#Test	PROGOL or GOLEM (Majority or Negative)		FEATURE ALONE		BANNAR	
		Covered	Uncovered	Covered	Uncovered	Covered	Uncovered
TCR	2,143	1540/1611	3/532	1540/1611	222/532	1570/1611	453/532
FEM	278	146/200	14/78	146/200	24/78	145/200	34/78
MUTA	188	100/109	54/79	100/109	54/79	102/109	55/79
KRK	10,000	3350/3356	6633/6644	3350/3356	6633/6644	3352/3356	6638/6644

3 to 222 and increased from 14 to 24 for the TCR and FEM datasets, respectively. This shows the effectiveness of our features for the classification of uncovered examples in multi-class problems. "FEATURE ALONE" did not help in increasing accuracy for examples covered by the rules nor for examples in two-class problems.

Comparing BANNAR to the method of "FEATURE ALONE", we can see that more uncovered-examples were correctly classified by BANNAR; the number of correctly classified uncovered-examples increased from 222 to 453 and increased from 24 to 34 for the TCR and FEM datasets, respectively. Another advantage of BANNAR over "FEATURE ALONE" was the higher accuracy for covered examples; e.g. the number of correctly classified covered-examples increased from 1540 to 1570 in the TCR dataset. All the above results show that about half of the improvement of BANNAR, which contains two main elements (features and neural networks), was due to the contribution of our features in the case of uncovered examples in multi-class problems. However, as "FEATURE ALONE" took each feature in a rule with equal significance, i.e. it simply counted the number of true-features for calculating *matchRatio*'s, it was unable to give important features higher weights than others. After neural networks were incorporated into BANNAR, more improvements were obtained as shown in the case of covered examples as well as in the case of uncovered examples.

3.7. Summary

In this section, we have demonstrated that BANNAR achieved good performance in classifying noisy or unseen data. The first set of experiments in Section 3.3 shows that BANNAR successfully increased the accuracy of the original rules obtained by PROGOL or GOLEM, especially on multi-class problems. After analyzing the improvements,

we found that the increased accuracy was significantly obtained in the case of data uncovered by the rules. These results show that BANNAR achieves its main objective that is aimed at increasing the accuracy of rules when no rule perfectly covers the unseen data.

The higher accuracy of BANNAR compared to PROGOL and GOLEM is clearly due to the fact that noisy data or unseen data are often not covered by the rules, and the use of only the majority or negative class does not perform well. The higher accuracy of BANNAR over the other learning systems, i.e. TILDE, 1BC, LINUS, is certainly due in part to the high accuracy of rules produced by PROGOL or GOLEM, and the approximate match of our method.

We next study if our method can improve the accuracy of rules in two-class problems with the presence of noise. The results in Section 3.4 confirmed the usefulness of our approximate match for increasing the accuracy of the overfitting rules. As shown by an example in Figure 3, the neural network was able to give higher weights to important features and give less attention to unimportant ones. These results demonstrated the ability of neural networks which is useful for the approximate match.

The approximate match is realized by two main elements: (1) our feature generation that enables partial match between the examples and the useful parts of rules which examine some properties of the examples, and (2) backpropagation neural networks that make use of features to enable more flexible match of rules with the examples. Though the usefulness of the approximate match was confirmed by the experiments on first-order datasets and the KRK noisy dataset, we want to know more about the contribution of each element to the increased accuracy. We then tried to give some evaluation of the contribution of these two elements by designing the additional experiments. The experimental results in Table VII show that about half of the improvement was due to the contribution of features (the simple algorithm "FEATURE ALONE" in the table). "FEATURE ALONE" significantly increased the accuracy of the original rules in the case of uncovered examples in multi-class problems. This validates the usefulness of features constructed by our method. The disadvantage of "FEATURE ALONE" is the lack of ability to give appropriate weights to features, as it simply counts the number of true-features and all features for determining the best matching rule. Neural networks, on the other hand, have ability to give higher weights to important features, and also play an important role in the improvement.

To summarize, the good performance of BANNAR is due to two main factors: (1) the high accuracy of rules produced by PROGOL or GOLEM, and (2) the approximate match of rules with unseen or noisy

data that can capture useful features and assign appropriate weights to the features by using our feature generation and backpropagation neural networks, respectively.

4. Results on Propositional Domains

This section presents the results of our method on propositional domains. We compare our method with C4.5 (Quinlan, 1993). For each dataset, we employ C4.5 to generate an unpruned decision tree and convert the tree to a propositional rule set. Having a rule set, we then build our BNN as described above. Our method can be thought of as *soft-pruning* of decision trees. By soft-pruning, we mean that a feature will not be completely pruned but will be given a low weight if it is not so important and a high weight if it is important.

To evaluate our soft-pruning method, we ran experiments to compare BANNAR with C4.5's unpruned and pruned trees. We also included the results of BANNAR using rules converted from C4.5's pruned trees, as we want to see how over-specific rules effect the accuracy of BANNAR. Twenty datasets of multi-class learning domains from the UCI repository (Merz et al., 1997) were used. In case of datasets where training and test sets were already provided, the results were evaluated on the given test sets. In the other datasets providing no test set, the results were averaged using 6-fold cross-validation[8]. Table VIII and Table IX report the summary of the datasets and the classification accuracies of BANNAR and C4.5.

The results show that BANNAR was more accurate than C4.5's unpruned trees in 12 datasets, and less in 4 datasets. Compared with C4.5's pruned trees, BANNAR achieved a higher accuracy in 11 datasets and lower in 8 datasets. The results were not as good as the ones for the first-order datasets. This is because C4.5 is a multi-class learner and more importantly it can effectively deal with noisy data by pruning the overfitting trees. However, if we include confidence levels in the comparison, we can see that BANNAR performs significantly better than C4.5's unpruned and pruned trees on 6 and 5 datasets, respectively. BANNAR performs significantly worse than C4.5's pruned trees only on one dataset. These results demonstrate the usefulness of our method in soft-pruning decision trees. The better results are due to more flexibility in pruning of our method: (1) converting a tree to rules before pruning will produce a non-leaf node (feature) of the tree in

[8] Compared to the first-order domains which require long time to train ILP systems, C4.5 ran fast in these propositional domains, and thus instead of 3-fold we used 6-fold cross-validation in the experiments.

Table VIII. The numbers of training data, test data and classes in the datasets tested in the experiments. 6CV denotes the experiment that uses six-fold cross-validation.

Data Set	#Train	#Test	#Classes	Data Set	#Train	#Test	#Classes
Allbp	2800	972	3	LED 17	2000	500	10
Allhyper	2800	972	5	Lymphography	148	6CV	4
Allhypo	2800	972	5	Primary-tumor	339	6CV	22
Allrep	2800	972	4	Satimage	4435	2000	6
Anneal	798	100	6	Segment	2310	6CV	7
Balance-scale	625	6CV	3	Shuttle	43500	14500	7
Glass	214	6CV	6	Soybean	307	376	19
Image	210	2100	7	Waveform	5000	6CV	3
Iris	150	6CV	3	Waveform+noise	5000	6CV	3
LED	2000	500	10	Wine	178	6CV	3

more than two rules, and thus this feature can be separately soft-pruned according to its participation in the rules, and (2) a feature that has some degree of significance for a rule will not be completely pruned, and will be assigned with an appropriate weight by the neural network.

Comparing the results of BANNAR with BANNAR using pruned trees, we can see that using pruned trees slightly reduced the average accuracy of BANNAR. Standard BANNAR significantly performed better than BANNAR using pruned trees on 4 datasets, and worse on 2 datasets. This shows that BANNAR performed better when it was provided with over-specific rules converted from C4.5's unpruned trees. Even when provided by rules converted from pruned trees, BANNAR still preserved its good performance.

5. Related and Future Work

There have been earlier learning systems which employ neural networks to improve the performance of symbolic learning. KBANN (Towell & Shavlik, 1994) and FONN (Botta et al., 1997) are the systems that use initial rules and training examples for learning neural networks. KBANN translates a propositional theory into a neural network, and uses training examples to refine the network. The results of KBANN show that its performance is better than methods which learn purely from examples. The main differences between BANNAR and KBANN are the first-order representation of the rules and the method of feature generation used in BANNAR. FONN translates first-order rules possibly including literals with new variables into a neural network. Its

Table IX. The percent accuracies of BANNAR and C4.5 on propositional domains. The second and third columns show the accuracies of C4.5's unpruned and C4.5's pruned trees, respectively. The accuracies of BANNAR are given in the fifth column, and those of BANNAR using pruned trees are given in the fourth column. The row "Won-loss-tied (BANNAR - XXX)" shows the performance comparison between BANNAR and the corresponding systems; where XXX is C4.5 or BANNAR using pruned trees. Superscripts denote confidence levels for the difference in accuracy between BANNAR and the corresponding system, using a one-tailed paired t test: $+$ or $-$ is 90%, $++$ or $--$ is 95%, $+++$ or $---$ is 99%; no superscripts denote confidence levels that are below 90%($+$ indicates higher accuracy of BANNAR, whereas $-$ indicates lower accuracy of BANNAR). Note that the highest accuracy for each data set is shown in bold face.

Data Set	C4.5 (Unpruned)	C4.5 (Pruned)	BANNAR+Pruned	BANNAR
Allbp	96.81	97.84	**97.94**	97.12
Allhyper	**98.87**	98.56	98.46	**98.87**
Allhypo	99.49	99.49	99.49	**99.69**
Allrep	98.77	**99.07**	**99.07**	98.97
Anneal	**97.00**	95.00	96.00	**97.00**
Balance-scale	69.76^{+++}	65.77^{+++}	$\mathbf{89.92}^{-}$	86.56
Glass	66.34	66.34	**69.11**	67.22
Image	89.43	91.00	$\mathbf{92.38}^{--}$	90.38
Iris	**95.33**	94.00	94.00	**95.33**
LED	**75.40**	75.20	74.80	74.60
LED 17	66.40	$\mathbf{75.20}^{---}$	65.00	63.80
Lymphography	73.70	78.38	**80.42**	77.72
Primary-tumor	**42.19**	41.32	33.93^{++}	38.66
Satimage	84.90^{++}	85.45	86.35	**86.80**
Segment	96.97	96.97	**97.01**	96.75
Shuttle	99.97^{+}	99.95^{++}	99.96^{++}	**99.99**
Soybean	85.64^{++}	86.70^{++}	86.44^{++}	**90.96**
Waveform	75.76^{+++}	75.82^{+++}	79.40^{++}	**80.30**
Waveform+noise	75.22^{+++}	75.30^{+++}	79.38	**79.64**
Wine	93.30	93.30	93.30	93.30
Average	84.06	84.53	85.62	85.68
Won-loss-tied (BANNAR - XXX)	12-4-4	11-8-1	10-9-1	

primary goal is to refine numerical literals of the rules. It first finds only bindings for new variables that satisfy non-numerical literals and uses these bindings for refining the numerical literals (Botta et al., 1997). The architecture of FONN is based on the Factorizable Radial Basis Function network, whereas BANNAR is based on the backpropagation neural network. Another difference between BANNAR and FONN is

our method of feature generation. Both KBANN and FONN as well as our system BANNAR have shown the similar results that the accuracy of rules can be increased by using neural networks.

Our method of feature generation is related to *clause templates* or *sketches* that are used as biases to define the hypothesis space in ILP systems (Bergadano & Gunetti, 1995; Tausend, 1994; Brazdil & Jorge, 1993). In MOBAL (Kietz & Wrobel, 1992), a *rule model*, which is a kind of clause templates, defines how background predicates are used to form a clause and defines the connection of variables via the predicates. The variable connection represents how variables in a clause are linked to other variables, and is similar to our closed and open chain features.

The feature generation (selection) has been addressed in earlier ILP systems, such as LINUS and 1BC. LINUS generates features for transforming a first-order example into a propositional representation in form of truth-value tuples described by the features. Depending on background knowledge which is used to generate features, the number of features can be very large. Lavrač et al. (Lavrač et al., 1999) describe a method for selecting only relevant features in order to reduce the hypothesis space searched by LINUS. The method is based on the *discriminating power* of a feature. Feature selection in LINUS and feature generation in BANNAR are employed for different purposes. The main purpose of feature selection in LINUS is for reducing the hypothesis space employed by an attribute-value learner, whereas our aim of feature generation is to generate a sequence of literals for partial matching with unseen examples. 1BC employs *structural predicates* and *properties* provided by users for generating *atomic* features. The features are then used to train the naive Bayes classifier for learning statistical information from training examples. The difference between our feature generation and that of 1BC is that our method does not require the user to give additional information for generating the features.

Both LINUS and 1BC generate features *directly* from background knowledge. However, our method needs rules obtained from PROGOL or GOLEM to generate features. This is a limitation of our method that cannot directly generate features from background knowledge. One of our future research plans is to extend the feature generation to a more powerful technique that can directly generate features from background knowledge. Another interesting research direction is to combine a naive Bayes classifier, like one in 1BC, with our method that generates features from previously learned rules, and use these features to learn statistical information for the classifier.

Another disadvantage of our method is that the learned neural networks are very difficult to understand to the user. We plan to study a

method that transfers the networks into a more understandable representation in the future.

Another direction for our future research is to investigate a more sophisticated method for evaluating the truth values of features, such as fuzzy logic. In the current work, if truth values of some literals of a feature are false, the truth value of the whole feature will be false. If we can assign a more suitable value, it may increase the classification accuracy.

There are some ILP systems that are able to learn multi-class concepts represented in first-order rules, such as RTL (Baroglio & Botta, 1996), ICL (De Raedt & Laer, 1995), MULT_ICN (Martin & Vrain, 1995). Nevertheless, it is still possible that some unseen data may not exactly match the rules produced by these systems. We believe that our method can also be applied to these systems.

6. Conclusions

We have proposed a method for approximate match of rules using neural networks. The main contribution of our work is the method of using feature generation and neural networks to increase the accuracy of first-order rules in case of multi-class problems or noisy domains where unseen data are often not covered by the rules. We have also proposed a novel technique to generate first-order features to be used for the approximate match of rules. Our method has been evaluated on four domains of first-order learning problems. The experimental results show that our method gives significant improvements over the use of the original rules. The improved results come from the combination of the ILP system and our system BANNAR. The ILP system produces rules that accurately classify the training data, and BANNAR makes the rules more flexible for partially matching with unseen or noisy data. We also applied BANNAR to approximate match of propositional rules converted from an unpruned decision tree. The results show that our method performs better than standard C4.5's pruned and unpruned trees.

Acknowledgements

We would like to thank Prabhas Chongstitvatana, Apinetr Unakul and Surapant Meknavin for comments on the paper. We are grateful to the anonymous referees for useful comments and suggestions that improved this work. This work is supported by the Thailand Research Fund. The

views and conclusions contained in this paper are those of the authors and the Thailand Research Fund is not necessarily of the same opinion.

References

Baroglio, C., & Botta, M. (1995). Multiple predicate learning with RTL. *Proceedings of the fourth Congress of the Italian Association for Artificial Intelligence, (AI*IA95)* (pp. 44-55). Florence, Italy.

Bergadano, F., & Gunetti, D. (1995). *Inductive Logic Programming: From Machine Learning to Software Engineering*. The MIT Press.

Blockeel, H., & Raedt, L. D. (1997). Experiments with top-down induction of logical decision trees. (Technical Report CW247). Dept. of Computer Science, K. U. Leuven.

Botta, M., Giordana, A., & Piola, R. (1997). FONN: Combining first order logic with connectionist learning. *Proceedings of the Fourteenth International Conference on Machine Learning* (pp.48–56). Morgan Kaufmann.

Brazdil, P., & Jorge, A. (1993). Exploiting algorithm sketches in ILP. *Proceedings of the Third International Workshop on Inductive Logic Programming*, Ljubljana, Slovenia, Jozef Stefan Institute.

Clark, P., & Niblett, T. (1989). The CN2 induction algorithm. *Machine Learning*, 3(4), 261–283.

Dolsak, B., Jezernik, A., & Bratko, I. (1994). A knowledge base for finite element mesh design, *Artificial Intelligence in Engineering*, 9, 19–27.

Dolsak, B., & Muggleton, S. (1992). The application of inductive logic programming to finite element mesh design. In S. Muggleton (Ed.), *Inductive Logic Programming* (pp. 453–472). Academic Press.

Džeroski, S., Schulze-Kremer, S., Heidtke, K. R., Siems, K., & Wettschereck, D. (1996). Applying ILP to diterpene structure elucidation from 13C NMR spectra. *Proceedings of the Sixth International Workshop on Inductive Logic Programming* (pp. 14–27). Springer.

Flach, P., & Lachiche, N. (1999). 1BC: A first-order Bayesian classifier. *Proceedings of the Ninth International Workshop on Inductive Logic Programming* (pp. 92–103). Lecture Notes in Artificial Intelligence (Vol. 1634). Springer-Verlag.

Fürnkranz, J. (1994). Fossil: A robust relational learner. *Proceedings of the Seventh European Conference on Machine Learning, ECML-94* (pp. 122–137). Springer-Verlag.

Kietz, J. U., & Wrobel, S. (1992). Controlling the complexity of learning in logic through syntactic and task-oriented models. In S. Muggleton (Ed.), *Inductive Logic Programming* (pp. 335–359). Academic Press.

Kijsirikul, B., & Sinthupinyo, S. (1999). Approximate ILP rules by backpropagation neural network: A result on Thai character recognition. *Proceedings of the Ninth International Workshop on Inductive Logic Programming* (pp. 162–173). Springer-Verlag.

Lavrač, N., & Džeroski, S. (1994). *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood.

Lavrač, N., Gamberger, D., & Jovanoski, V. (1999). A study of relevance for learning in deductive databases. *Journal of Logic Programming*, 40, 215–249.

Mahoney, J., & Mooney, R. (1994). Comparing methods for refining certainty-factor rule-bases. *Proceedings of the Eleventh International Workshop on Machine Learning*, Ruthers University, NJ.

Martin,L., & Vrain, C. (1995). MULT_ICN: An empirical multiple predicate learner. *Proceedings of the Fifth International Workshop on Inductive Logic Programming* (pp. 129–144).

Merz, C. J., Murphy, P. M., & Aha, D. W. (1997). UCI repository of machine learning databases. Department of Information and Computer Science, University of California, Irvine, CA. `http://www.ics.uci.edu/~mlearn/MLRepository.html`.

Muggleton, S. (1991). Inductive logic programming, *New Generation Computing*, 8(4), 295–318.

Muggleton, S. (1995). Inverse entailment and PROGOL. *New Generation Computing*, 13, 245–286.

Muggleton, S., Bain, M., Hayes-Michie, J., & Michie, D. (1989). An experimental comparison of human and machine learning algorithms. *Proceedings of the Sixth International Workshop on Machine Learning* (pp. 113–118). Morgan Kaufmann.

Muggleton, S., & Feng, C. (1990). Efficient induction of logic programs. *Proceedings of the First Conference on Algorithmic Learning Theory* (pp. 368–381). Ohmsha, Tokyo.

Muggleton, S., & Raedt, L. D. (1994). Inductive logic programming: Theory and methods, *Journal of Logic Programming*, 19:20, 629–679.

Nilsson, N. J. (1980). *Principles of Artificial Intelligence*. Tioga, Palo Alto, CA.

Plotkin, G. D. (1970). A note on inductive generalization. In B. Meltzer, & D. Michie (Eds.), *Machine Intelligence* (Vol. 5). Elsevier North-Holland, New York.

Quinlan, J. R. (1990). Learning logical definitions from relations. *Machine Learning*, 5(3), 239–266.

Quinlan, J. R. (1993). *C4.5:Programs for Machine Learning*, Morgan Kaufmann, San Mateo, CA.

Raedt, L. D., & Laer, W. V. (1995). Inductive constraint logic. *Proceedings of the Fifth Workshop on Algorithmic Learning Theory* (pp. 80–94). Lecture Notes in Artificial Intelligence (Vol. 997). Springer-Verlag.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning internal representations by error propagation. In D. E. Rumelhart, & J. L. McClelland (Eds.), *Parallel distributed processing* (Vol. 1). Cambridge, MA:MIT Press.

Srinivasan, A., Muggleton, S., Sternberg, M. J. E., & King, R. D. (1996). Theories for mutagenicity: A study in first-order and feature-based induction. *Artificial Intelligence*, 85, 277–299.

Tausend, B. (1994). Representing biases for inductive logic programming. *Proceedings of European Conferences on Machine Learning, ECML-94* (pp.427–430). Springer-Verlag.

Towell, G. G., & Shavlik, J. W. (1994). Knowledge-based artificial neural networks. *Artificial Intelligence*, 70(4), 119–116.