A Relevancy Filter for Constructive Induction

Nada Lavrač, Jožef Stefan Institute
Dragan Gamberger, Rudjer Bošković Institute
Peter Turney, Institute for Information Technology, National Research Council Canada

THE REDUCE ALGORITHM ELIMINATES IRRELEVANT TERMS IN LEARNING TASKS. IN THIS CASE STUDY, THE AUTHORS USE REDUCE TO PREPROCESS DATA FOR A HYBRID GENETIC ALGORITHM RL-ICET.

SOME MACHINE-LEARNING ALGOrithms enable the learner to extend its vocabulary with new terms if, for a given a set of training examples, the learner's vocabulary is too restricted to solve the learning task. We propose a filter that selects potentially relevant terms from the set of constructed terms and eliminates terms that are irrelevant for the learning task. Restricting constructive induction (or predicate invention) to relevant terms allows a much larger explored space of constructed terms. The elimination of irrelevant terms is especially well-suited for learners of large time or space complexity, such as genetic algorithms and artificial neural networks.

To illustrate our approach to feature construction and irrelevant feature elimination, we applied our proposed relevancy filter to the 20- and 24-trains East-West Challenge problems.[1] The experiments show that the performance of a hybrid genetic algorithm, RL-ICET,[2] improved significantly when we applied the relevancy filter while preprocessing the dataset.

Our relevancy filter

We can view inductive concept learning as a process of searching a space of concept descriptions or hypotheses. The language bias determines the space of hypotheses to be searched. Syntactic restrictions of the hypothesis language and the vocabulary of terms in the language, as well as the vocabulary of functions and relations defined in the background knowledge, determine the language bias.

Let's consider a propositional learning task where, given a fixed set of attributes, training examples are represented by tuples of *features* (attribute values). If the given vocabulary is too restricted for the learning task, *constructive induction*[3] can extend the hypothesis language with new terms, automatically constructed from the terms in the learner's vocabulary or generated based on the functions and relations defined in the background knowledge. The problem then becomes how to select, from the set of constructed terms, only those terms that are relevant for the given task.

First-order learning of relational descriptions, or inductive logic programming,[4] assumes a given set of training examples, represented by a relational table, and background knowledge, represented either extensionally (in the form of relational tables) or intentionally (in the form of rules). If the given language bias is too restricted for the learning task, *predicate invention*[5] can invent definitions of new predicates from the definitions of predicates in the background knowledge, thus causing bias shift. Alternatively, bias shift can occur by allowing the learner to switch its search to a hypothesis space defined by a syntactically more expressive hypothesis language.

Constructive induction and predicate invention can be invaluable for the success of learning. Expansion of the hypothesis language can, however, decrease the learner's

1094-7167/98/$10.00 © 1998 IEEE

performance, particularly in terms of efficiency. But, despite the extended hypothesis language, there is no guarantee that the new vocabulary will help the learner to induce a better solution, because the new terms might be irrelevant for the task at hand.

To prevent uncontrolled expansion of the hypothesis language, we've developed a filter that distinguishes the potentially relevant terms from those that are irrelevant for the learning task. We can use the relevancy filter to eliminate the irrelevant terms while preprocessing the set of training examples. By biasing constructive induction (or predicate invention) to relevant terms only, the explored space of constructed terms can be larger, which increases the chance of successfully solving the learning task.

Our filter is directly applicable to two-class propositional learning problems described with binary-valued terms, and to those inductive logic programming tasks for which a Linus-like transformation approach is applicable.[4] When applying the filter to a multiclass learning problem, we must transform the problem to a two-class learning problem; the transformation can be done the same way as in the well-known covering algorithms AQ and CN2. As this article describes, multivalued attributes must be transformed into a binary-valued literal form.

The relevance of literals and features

Let's consider a two-class learning problem where the training set E consists of positive and negative examples of a concept ($E = P \cup N$) and examples $e \in E$ are tuples of truth values of terms in a hypothesis language. L denotes the set of all terms, called *literals*.

We'll represent the training set E as a table whose rows correspond to training examples and columns correspond to literals. An element in the table has the value `true` when the example satisfies the condition (literal) in a column of the table; otherwise its value is `false`.

How to achieve the required propositional representation. If the training set does not have the form of truth-value tuples, a preprocessor has to transform the training set to transform it to this form. For attribute-value learning, the transformation procedure is based on analysis of the values of examples in the training set. For each attribute A_i, let v_{ix} ($x = 1..k_{ip}$) be the k_{ip} different values of the attribute that appear in the positive examples and let w_{iy} ($y = 1..k_{in}$) be the k_{in} different values appearing in the negative examples. The transformation results in this set of literals L:

- For discrete attributes A_i, literals of the form $A_i = v_{ix}$ and $A_i \neq w_{iy}$ are generated.
- For continuous attributes A_i, literals of the form $A_i \leq (v_{ix} + w_{iy})/2$ are created for all neighboring value pairs (v_{ix}, w_{iy}), and literals $A_i > (v_{ix} + w_{iy})/2$ for all neighboring pairs (w_{iy}, v_{ix}).
- For integer valued attributes A_i, literals are generated as if A_i were both discrete and continuous, resulting in literals of four different forms: $A_i \leq (v_{ix} + w_{iy})/2$, $A_i > (v_i + w_{iy})/2$, $A_i = v_{ix}$, and $A_i \neq w_{iy}$.

TO PREVENT UNCONTROLLED EXPANSION OF THE HYPOTHESIS LANGUAGE, WE'VE DEVELOPED A FILTER THAT DISTINGUISHES THE POTENTIALLY RELEVANT TERMS FROM THOSE THAT ARE IRRELEVANT FOR THE LEARNING TASK.

Construction of new terms. We assume that new terms also have the form of literals, which have `true` or `false` values for the given training examples. Suppose that attributes A_i describe the original training set. Constructive induction can create new terms such as:

- literals that test the relations $A_i = A_j$, $A_i \neq A_j$ for attributes of the same type (the same sets of values or continuous attributes),
- literals introducing internal disjunctions $A_i = [v_{ik} \vee v_{il}]$ or intervals $A_i \in [v_{ik}, v_{il}]$,
- conjunctions of features $(A_i = v_{ik}) \wedge (A_j = v_{jl})$,
- literals that test values of functions defined in the background knowledge, such as $f(A_i, A_j, \ldots) \leq v$, and
- literals using the relations defined in the background knowledge $r(A_i, A_j)$.

An obvious way to construct new terms is also by generating negations of literals (the so-called *negative literals*): for every $r(A_i, A_j)$ construct a literal $\neg r(A_i, A_j)$.

The Linus learning system proposes a general procedure for constructing terms based on the information in the background knowledge.[4] We can use that procedure to construct new terms in attribute-value learning, as well as in first-order learning (inductive logic programming), where we can solve the relation learning tasks by using Linus's restricted hypothesis language of constrained nonrecursive clauses with typed variables.

For example, consider a relation-learning task, where the training set consists of examples of the target relation *daughter*(A_1, A_2) and the background knowledge consists of definitions of a unary relation *female*, a binary relation *parent*, and the equality relation (=). The transformation of training examples results in a table whose rows are truth-value tuples corresponding to individual training examples and whose columns correspond to literals, constructed by the appropriate variabilization of background-knowledge predicates: *female*(A_1), *female*(A_2), *parent*(A_1, A_2), *parent*(A_2, A_1), *parent*(A_1, A_1), *parent*(A_2, A_2), and $A_1 = A_2$.

The variabilization of constructed literals is restricted to the use of variables A_1 and A_2, because in constrained clauses only the variables appearing in an induced clause's head may appear in the literals of the clause's body. Thus, for the target relation *daughter*(A_1, A_2), the constructed literals may only use A_1 and A_2 in the arguments, whereas literals introducing new variables, such as *parent*(A_1, Z), are disregarded. Despite this limitation of the hypothesis language, complexity analysis of the Linus transformation approach reveals that the number of constructed literals is linear in the number of background-knowledge predicates and exponential in the number of arguments (of different types) of the background-knowledge predicates. Therefore, eliminating irrelevant literals constructed by the Linus transformation procedure is invaluable for the learner's success, when the number and arity of predicates in the background knowledge is large.

Unlike Linus's proposed general procedure, our case study presents a special-purpose procedure for constructing new terms in the

East-West challenge problem, where the construction of new terms turns out to be the key to the learning task's successful solution.

The *p/n* pairs of examples. Let a truth-value table represent the set of training examples E, where columns correspond to the set of (positive and negative) literals L and rows are truth-value tuples of literals, representing training examples $e \in E$. The table has two parts, P and N, where P are the positive examples and N are the negative examples. Let $P \cup N$ denote the truth-value table E.

We use these definitions and notation:[6]

- A *p/n pair* is a pair of training examples where $p \in P$ and $n \in N$.
- Literal $l \in L$ *covers* a p/n pair if in column l of the table of training examples E the positive example p has the value `true` and the negative example n has the value `false`.
- $E(l)$ denotes the set of all p/n pairs covered by literal l.
- Literal l *covers* literal l' if $E(l') \subseteq E(l)$.

The relevance of literals. Consider a simple learning problem with five training examples forming example set E: three positive $P = \{p_1, p_2, p_3\}$ and two negative $N = \{n_1, n_2\}$. Examples are described by the truth-values of literals $l_i \in L$. Table 1 shows just some of the truth-values of E.

To understand the meaning of p/n pairs and the notions of coverage and relevance of literals, consider just three literals: l_2, l_4, and l_8. Literal l_2 in Table 1 appears to be relevant for the formation of an inductive hypothesis because it is true for a positive example and false for both negative examples; a hypothesis constructed from l_2 only would cover the positive example p_2 and would not cover the negative examples n_1 and n_2. Literal l_2 thus appears to be a reasonable ingredient of an inductive hypothesis aimed at covering all the positive examples and none of the negative examples.

Table 1. Coverage of literals and *p/n* pairs.

Examples					Literals			
		… …	l_2	… …	l_4	… … …	l_8	… …
	p_1							
P	p_2		true		true		false	
	p_3							
	n_1		false		false		true	
N	n_2		false		true		true	

Literal l_2 covers two p/n pairs: $E(l_2) = \{p_2/n_1, p_2/n_2\}$. Literal l_8 is inappropriate for constructing a hypothesis, because it does not cover any p/n pair: $E(l_8) = \emptyset$. Literal l_4 seems to be less relevant than l_2 and more relevant than l_8 because it covers one p/n pair: $E(l_4) = \{p_2/n_1\}$. Literal l_2 covers l_4 and l_8, and literal l_4 covers l_8, because $E(l_8) \subseteq E(l_4) \subseteq E(l_2)$.

The example in Table 1 lets us reach this intuition: the more p/n pairs a literal covers, the more relevant it is for hypothesis formation. We can formalize this intuition in this definition:

> Literal l' is *irrelevant* if a literal $l \in L$ exists such that l covers l' ($E(l') \subseteq E(l)$).

In other words, literal l' is irrelevant if it covers a subset of p/n pairs covered by some other literal $l \in L$.

Let us now assume that literals are assigned costs. Let $c(l)$ denote the cost of literal $l \in L$. We need to modify the definition of irrelevance if we want to take into the account the costs of literals:

> Literal l' is *irrelevant* if a literal $l \in L$ exists such that l covers l' ($E(l') \subseteq E(l)$) and the cost of l is lower than the cost of l' ($c(l) \leq c(l')$).

In our case study, cost is a measure of complexity: the more syntactically complex the literal, the higher its cost.

A filter for cost-sensitive elimination of irrelevant literals. If a constructive induction procedure generates irrelevant literals, they can be detected and eliminated before entering the learning process. Figure 1 presents a cost-sensitive algorithm for irrelevant literal elimination. The algorithm assumes a set of training examples E described by an initial set of relevant literals L, a set of constructed literals L_{New}, and the assignment of literal costs $c(l)$. If no costs are assigned, we assume for all $l \in L \cup L_{New}$: $c(l) = 1$.

Implementation issues. We can efficiently implement the relevancy filter using simple bitstring manipulation on the table of training examples E. For this purpose, we transform the table E into E_t:

$\forall p \in P$: replace `true` by 1 and `false` by 0
$\forall n \in N$: replace `false` by 1 and `true` by 0

In this representation, examples $e \in E_t$ and literals $l \in L$ are bitstrings. Coverage can now be checked by set inclusion. Recall that literal l covers literal l' if $E(l') \subseteq E(l)$. Thus, if l' has the value 1 only in (some of) those rows in which l has 1 and in no other rows, l' is irrelevant and can be eliminated.

The relevancy filter assumes that the initial literal set consists of relevant literals. Alternatively, we can consider the entire set $L \cup L_{New}$ as a unique literal set to be checked for the relevance of its elements; we used this variant of the relevancy filter in our experiments. This algorithm, called Reduce,[7] was initially developed by Dragan Gamberger and implemented in the Inductive Learning by Logic Minimization (ILLM) algorithm for inductive-concept learning. Reduce first eliminates all the useless literals l_i (literals that are `false` for all $p \in P$ and those that are `true` for all $n \in N$) and continues by eliminating other irrelevant literals in turn.

Relevance of features. The term *feature* denotes a positive literal—for example, $A_i = v$, $A_j \leq w$, $r(A_i, A_j)$. In the hypothesis language, the existence of one such feature implies the existence of two complementary literals: a positive and a negative literal. Suppose that we consider the feature `Color = black` and that the attribute `Color` has three possible values: `black`, `white`, and `red`. Because each feature implies the existence of two literals, the necessary and sufficient condition that a feature can be eliminated as irrelevant is that both of its literals `Color = black` and `Color ≠ black` (that is, $\neg$(`Color = black`)) are irrelevant. This statement directly implies the procedure taken in our experiment. First, we convert the starting feature vector to the corresponding literal vector, which has twice as many elements. Next, we eliminate the irrelevant literals and then construct the reduced set of features, which includes all the features that have at least one of their literals in the reduced literal vector.

```
Given: costs $c(l)$ of literals in $L \cup L_{New}$
Input: $L$ - initial set of relevant literals, $L_{New}$ - new literals,
    $E = P \cup N$ - table of positive and negative
    examples, consisting of truth-value tuples of $L$

    for $\forall\ l_i \in L_{New}$ do
        for $\forall p \in P$ and $\forall n \in N$ evaluate $l_i$ as true or false
        if $\forall p \in P\ l_i$ has value false then $l_i$ is irrelevant
        if $\forall n \in N\ l_i$ has value true then $l_i$ is irrelevant
        if $l_i$ is covered by any $l_j \in L$ for which $c(l_j) \leq$
        $c(l_i)$ then $l_i$ is irrelevant
        else $L \leftarrow L \cup \{l_i\}$, and add column of truth values of
        $l_i$ to table $E = P \cup N$
    endfor

Output: $L$ - extended set of relevant literals, $E = P \cup N$ - extended
     table of positive and negative examples, consisting of
     truth-value tuples of $L$
```

Figure 1. The relevancy filter: an algorithm for cost-sensitive elimination of irrelevant literals.

However, direct detection of irrelevant features (without conversion to and from the literal form) is impossible except in the trivial case where two or more features have identical columns in the E_t table. Only in this case does a feature f exist whose literals f and $\neg f$ cover both literals g and $\neg g$ of some other feature. Generally, if a literal of feature f covers some literal of feature g, the other literal of feature f does not cover the other literal of feature g. But sometimes a literal of some other feature h covers this other literal of feature g. Therefore, although no such feature f covers both literals of feature g, feature g can still turn out to be irrelevant.

This analysis supports the approach, used in our study, in which the input to the learner is tuples of truth values of positive and negative literals, rather than feature vectors, which are used in most standard approaches to rule induction.

Utility study: the East-West Challenge

Donald Michie and his colleagues issued a challenge to the international computing community to discover low size-complexity Prolog programs for classifying trains as eastbound or westbound.[1] The challenge was inspired by a problem posed by Ryszard Michalski and J.B. Larson, in 1977, where the task was to generate rules that will classify trains as east- or westbound. Figure 2 illustrates the original problem.

Michie's original challenge included three separate tasks.[1] He later issued a second challenge, involving a fourth task. Our experiments involve the first and fourth tasks. The first task had 20 trains, 10 eastbound and 10 westbound, whereas the fourth task involved 24 trains, 12 eastbound and 12 westbound. The challenge in these tasks was to discover the simplest rule for distinguishing the eastbound and westbound trains.

For both tasks, the winner was decided by representing the rule as a Prolog program and measuring its size-complexity. The size-complexity was calculated as the sum of the numbers of clause occurrences, term occurrences, and atom occurrences. The performance on these two tasks was judged by size-complexity, not by accuracy on independent testing data (there were no independent testing data). All rules competing in the challenge were required to achieve 100% accuracy on the data.

RL-ICET. ICET is a cost-sensitive algorithm, a hybrid of a genetic algorithm (Grefenstette's Genesis) and a decision-tree induction algorithm (Quinlan's C4.5), designed to generate low-cost decision trees. ICET performs a two-tiered search. On the bottom tier, C4.5 searches through the space of decision trees. On the top tier, Genesis searches through the space of biases.

ICET takes feature vectors as input and generates decision trees as output, using its C4.5 component. We modified the C4.5 component from Quinlan's original design, so that its learning bias can be controlled by a vector of real-valued parameters, called a *bias vector*. ICET's Genesis component searches in the space of bias vectors for a bias that optimizes the performance of the C4.5 component, according to a given performance measure. ICET uses a performance measure that is sensitive to both the cost of features (the cost of acquiring information about an element in a feature vector) and the cost of classification errors (the cost of mistaken classifications made by the output decision tree).[2]

Although ICET takes feature vectors as input and generates decision trees as output, the East-West Challenge involves input data in the form of Prolog relations and output theories in the form of Prolog programs. For the East-West Challenge, we extended ICET to handle Prolog input. This algorithm is called RL-ICET (Relational Learning with ICET).[2]

RL-ICET is similar to the Linus learning system, as they both use a three-part learning strategy:[4]

(1) A preprocessor translates the Prolog relations and predicates into a feature-vector format. We designed the pre-

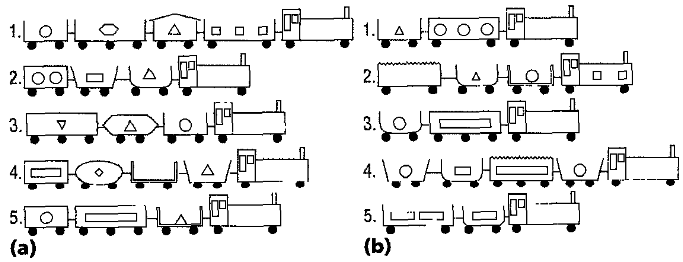
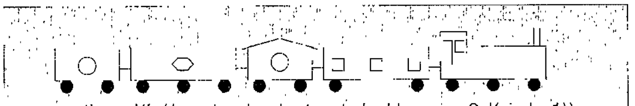


Figure 2. The original 10-train East-West Challenge: (a) trains going east, (b) trains going west.



```
eastbound([c(1, rectangle; short. not_double, none, 2, l(circle, 1)),
c(2, rectangle, long, nol_double, none, 3, l(hexagon, 1)),
c(3, reciangle, short. noi_double, peaked, 2, l(triangle, 1)),
c(4. rectangle, long, not_double, none, 2. l(rectangle, 3))]).
```

Figure 3. A train and its Prolog clause representation.

processor in RL-ICET especially for the East-West Challenge; Linus has a general-purpose preprocessor.

(2) An attribute-value learner applies a decision-tree induction algorithm (ICET) to the feature vectors. Each feature is assigned a cost, based on the size of the fragment of Prolog code that represents the corresponding predicate or relation. A low-cost decision tree corresponds (roughly) to a Prolog program that has a low size-complexity. When it searches for a low-cost decision tree, ICET is in effect searching for a low size-complexity Prolog program.

(3) A postprocessor translates the decision tree into a Prolog program. RL-ICET performs postprocessing manually; Linus does it automatically.

Feature construction in RL-ICET. Much of RL-ICET's success in the East-West Challenge tasks are attributable to its preprocessor. The data about each train in the East-West Challenge were represented using Prolog. For example, the Prolog clause shown in Figure 3 represents the train also shown in the figure.

We used a simple Prolog program to convert the relatively compact Prolog description into a feature-vector format (tuples of truth-values of features) for use in decision-tree induction. This produced rather large feature vectors of 1,199 elements. These large vectors ensure that all the features that are potentially interesting for the final solution are made available for ICET.

To construct features, we started with 28 predicates that apply to the cars in a train, such as `ellipse(C)`, which is true when the car C has an elliptical shape. For each of these 28 predicates, we defined a corresponding feature. All features were defined for whole trains, rather than single cars, because the problem is to classify trains. The feature `ellipse`, for example, is `true` when a given train has a car with an elliptical shape. Otherwise `ellipse` is `false`.

We then defined features by forming all possible unordered pairs of the original 28 predicates. For example, the feature `ellipse_triangle_load` is `true` when a given train has a car with an elliptical shape that is carrying a triangle load, and `false` otherwise. For a given train, the features `ellipse` and `triangle_load` can be `true`, while the feature `ellipse_triangle_load` is `false`, because it is only `true` when the train has a car that is both elliptical and carrying a triangle load.

Table 2. Some features and their costs.

FEATURE	PROLOG FRAGMENT	COST
ellipse	has_car(T, C), ellipse(C).	5
short_closed	has_car(T, C), short(C), closed(C).	7
train_4	len1(T, 4).	3
train_hexagon	has_load1(T, hexagon).	3
ellipse_peaked_roof	has_car(T, C), ellipse(C), arg(5, C, peaked).	9
u_shaped_no_load	has_car(T, C), u_shaped(C), has_load(C, 0).	8
rectangle_load_infront _jagged_roof	infront(T, C1, C2), has_load0(C1, rectangle), arg(5, C2, jagged).	11

Next, we defined features by forming all possible ordered pairs of the original 28 predicates, using the relation `infront(T, C1, C2)`. For example, the feature `u_shape_front_peaked_roof` is `true` when the train has a U-shaped car in front of a car with a peaked roof, and `false` otherwise. Finally, we added nine more predicates that apply to the train as a whole, such as `train_4`, which is `true` when the train has exactly four cars. Thus a train is represented by a feature vector, where every feature is either `true` or `false`.

We assigned each feature a cost, based on the complexity of the fragment of Prolog code required to represent that feature. Recall that we define the complexity of a Prolog program as a sum of the numbers of occurrences, term, and atom occurrences. Table 2 shows some constructed features and their costs.

A train's feature vector does not capture all the information in the original Prolog representation. For example, we could also define features by combining all possible unordered triples of the 28 predicates. However, these features would likely be less useful, because they are so specific that they will only rarely be `true`. If the target concept should happen to be a triple of predicates, it could be closely approximated by the conjunction of the three pairs of predicates that are subsets of the triple.

This kind of translation to feature-vector representation could apply to many other types of structured objects. For example, consider the problem of classifying a set of documents. The keywords in a document are analogous to the cars in a train. The distance between keywords or the order of keywords in a document might be useful when classifying the document, just as the `infront` relation might be useful when classifying trains.

Feature elimination. The objective of our experiments was to show the utility of the Reduce literal-elimination algorithm. Our approach lets us generate many different features, which will surely include all significant ones, and then, before using an inductive learner, eliminate all irrelevant features to keep the computation as effective as possible. We performed two separate experiments for the 20- and 24-trains problems. In both experiments, we used the RL-ICET preprocessor to generate the appropriate features and transform the training examples into a feature-vector format. This resulted in two training sets of 20 and 24 examples each.

To apply the Reduce algorithm, we first

converted the 1,199-element starting feature vector to the corresponding literal vector, which has twice as many elements, containing 1,199 features generated by the RL-ICET preprocessor (positive literals) as well as their negated counterparts (1,199 negative literals). After that, we eliminated the irrelevant literals and, in the third phase, constructed the reduced set of features, which includes all features that have at least one of their literals in the reduced literal set.

We tested Reduce's utility as follows. First, we performed 10 runs of the ICET algorithm on the set of training examples with 1,199 features. Then we performed 10 runs of ICET on the training examples with the reduced set of features selected by Reduce.

Table 3 summarizes the results. The table compares the average results of 10 runs of RL-ICET with respect to the costs of decision trees and execution times. The ICET algorithm's stochastic nature required us to use 10 runs: each time it runs, it yields a different result (assuming that the random number seed is changed). If we compared one single run of ICET on 1,199 features to one run of ICET on the reduced feature set, the outcome of the comparison could be due to chance. All trials are independent of each other. (For example, the results of trial 4 should not be compared to the results of trial 14.) Only the average results are relevant for the comparison.

In the RL-ICET experiments, we measured the performance by the cost of the decision trees induced by ICET, as well as the complexity of the Prolog programs after the RL-ICET transformation of decision trees into the Prolog program form.[2] In Table 3, we skip the latter, because the transformation into the Prolog form is currently manual and suboptimal, which means that a tree with the lowest cost found by ICET is not necessarily transformed into a Prolog program with the lowest complexity.

Results of the 20-trains experiment. With the 20-train data, Reduce cut the original set of 1,199 features to 86 features, thus reducing the complexity of the learning problem to about 7% (86/1,199) of the initial problem.

The results show that the efficiency of learning significantly increased. In the initial problem with 1,199 features, the average time per experiment was approximately 2 hours and 17 minutes; in the 86-feature reduced problem setting, the average time was approximately 12 minutes. The difference between times t_1 and t_2 is significant. This shows the utility of literal reduction for genetic algorithms, which are typically expensive in terms of CPU time.

The average cost of descriptions induced from the 86-feature set decreased (from 20 to 18.6), but the difference between decision tree costs c_1 and c_2 is not significant. The variance (and the standard deviation) of the costs was also decreased: the costs of the decision trees generated from 1,199 features vary more than the costs of the trees generated from 86 features: $var(c_1) = 1.6$ ($sd(c_1) = 1.3$) and $var(c_2) = 5.1$ ($sd(c_2) = 2.3$).

Results of the 24 trains experiment. In this experiment, Reduce decreased the number of features from 1,199 to 116, thus reducing the learning problem's complexity to about 10% (116/1,199) of the initial problem. Here, too, the efficiency of learning significantly increased. In the initial problem with 1,199 features, the average time per experiment was nearly two hours; in the 116-feature reduced problem setting, the average time was approximately 14 minutes. The difference between times t_1 and t_2 is significant.

The average cost of the decision trees induced from the 116-feature set also decreased. The difference between decision tree costs c_1 and c_2 is significant (at the 99.99% confidence level). Our hypothesis that variance (and standard deviation) of the output of RL-ICET can be reduced is only weakly supported, because the inequality of variance is insignificant: $var(c_1) = 4.8$ ($sd(c_1) = 2.2$) and $var(c_2) = 5.2$ ($sd(c_2) = 2.3$).

OUR AIM IN THIS WORK IS TO contribute to a better understanding of relevance for inductive-concept learning. Our case study shows that the construction of appropriate features can be crucial for the success of learning, and that cost-sensitive elimination of irrelevant features can substantially improve learning efficiency and reduce the costs of induced hypotheses.

The case study deals with an exact, noise-free problem in which we assume that the learning goal is to find a consistent and complete concept description. The choice of a simplified noise-free setting enabled a clear presentation of the notions underlying the

Table 3. Results of the experiments.

20 Trains						24 Trains					
86 Features			1,199 Features			116 Features			1,199 Features		
Trial	Time	Cost	Trial	Time	Cost	Trial	Time	Cost	Trial	Time	Cost
	t_1	c_1		t_2	c_2		t_1	c_1		t_2	c_2
1	11:05	18	11	2:21:32	24	1	14:35	20	11	1:54:15	27
2	11:19	21	12	2:21:34	21	2	14:26	18	12	1:55:29	21
3	12:55	18	13	2:19:15	20	3	14:59	18	13	2:00:25	26
4	11:35	18	14	2:19:32	20	4	14:17	21	14	1:56:31	25
5	15:16	18	15	2:16:20	18	5	13:32	18	15	1:56:47	25
6	11:35	18	16	2:23:52	22	6	13:31	22	16	1:57:14	24
7	11:32	18	17	2:24:09	21	7	14:29	18	17	1:56:52	28
8	11:38	18	18	2:18:41	16	8	13:54	23	18	1:56:33	23
9	11:28	18	19	2:16:58	18	9	13:51	23	19	1:49:08	27
10	11:18	21	20	2:23:09	20	10	14:30	18	20	1:47:46	28
Sum	119:41	186	Sum	23:25:02	200	Sum	2:22:04	199	Sum	19:11:00	254
Mean	11:57	18.6	Mean	2:16:54	20	Mean	14:12	19.9	Mean	1:55:05	25.4

implemented relevancy filter. Real-life problems require a more complex setting, involving the use of our algorithm for noise detection and elimination. Based on the notion of p/n pairs, the algorithm first heuristically evaluates the minimal number of literals needed to construct a complete and consistent hypothesis, and then detects in the training set the potentially noisy examples as those whose elimination will decrease the number of literals needed for hypothesis generation.[7]

Because RL-ICET originally required about two hours to process the 20- or 24-trains problems, the RL-ICET approach to inductive logic programming did not seem able to scale up to larger problems, with hundreds or thousands of instances.[2] However, with Reduce as a preprocessor, RL-ICET now requires under 15 minutes for the 20- or 24-trains problems. Reduce itself runs in seconds. The combination of Reduce and RL-ICET likely can scale up to much larger problems than were previously practical for RL-ICET alone. ■

Acknowledgments

This research was supported by the Slovenian Ministry of Science and Technology, the Croatian Ministry of Science, the National Research Council of Canada, and ESPRIT Project 20237, Inductive Logic Programming 2. We're grateful to Donald Michie for stimulating this work.

References

1. D. Michie et al., "To the International Computing Community: A New East-West Challenge," Oxford Univ. Computing Laboratory, Oxford, UK, 1994; ftp://ftp.comlab.ox.ac.uk/pub/Packages/ILP/Trains/trains.tar.Z.

2. P. Turney, "Low Size-Complexity Inductive Logic Programming: The East-West Challenge as a Problem in Cost-Sensitive Classification," in *Advances in Inductive Logic Programming*, IOS Press, Amsterdam, 1996, pp. 308–321.

3. R.S. Michalski, "A Theory and Methodology of Inductive Learning," in *Machine Learning: An Artificial Intelligence Approach*, R. Michalski, J. Carbonell, and T. Mitchell, eds., Tioga, Palo Alto, Calif., 1983, pp. 83–134.

4. N. Lavrač, and S. Džeroski, *Inductive Logic Programming: Techniques and Applications*, Ellis Horwood, Hemel Hempstead, UK, 1994.

5. I. Stahl, "Predicate Invention in Inductive Logic Programming," in *Advances in Inductive Logic Programming*, L. De Raedt, ed., IOS Press, 1996.

6. N. Lavrač, D. Gamberger, and P. Turney, "Cost-Sensitive Feature Reduction Applied to a Hybrid Genetic Algorithm," *Proc. Seventh Int'l Workshop an Algorithmic Learning Theory (ALT-96)*, Springer-Verlag, Berlin, 1996, pp. 127–134.

7. D. Gamberger, N. Lavrač, and S. Džeroski, "Noise Elimination in Inductive Concept Learning: A Case Study in Medical Diagnosis," *Proc. Seventh Int'l Workshop on Algorithmic Learning Theory*, Springer, 1996, pp. 199–212.

Nada Lavrač is a senior research associate at the Department of Intelligent Systems, J. Stefan Institute, Ljubljana, Slovenia, and a visiting professor at the Klagenfurt University, Austria. Her main research interest is machine learning, in particular inductive logic programming and intelligent data analysis in medicine. She received a BSc in technical mathematics and MSc in computer science from Ljubljana University and a PhD in technical sciences from Maribor University, Slovenia. She is coauthor of *KARDIO: A Study in Deep and Qualitative Knowledge for Expert Systems* (MIT Press, 1989) and *Inductive Logic Programming: Techniques and Applications* (Ellis Horwood, 1993), and coeditor of *Intelligent Data Analysis in Medicine and Pharmacology* (Kluwer, 1997). Contact her at the J. Stefan Inst., Jamova 39, 1000 Ljubljana, Slovenia; nada.lavrac@ijs.si.

Dragan Gamberger is a research assistant at the Rudjer Boskovic Institute in Zagreb, Croatia. His research interests include the application of machine learning, inductive learning by logic minimization, knowledge discovery in databases, knowledge representation, noise elimination, Occam's razor, and predicate invention. He received a BSc in electrical engineering, an MSc in electrical engineering, and a PhD in computer science, all from the University of Zagreb. Contact him at the Rudjer Boskovic Inst., Dept. of Physics, Information Systems Lab, Bijenicka 54, 10,000 Zagreb, Croatia; gamber@faust.irb.hr.

Peter Turney is a research officer in the Interactive Information Group of Canada's National Research Council. His research interests include the application of machine learning to industrial applications, information retrieval, evolutionary computation, machine learning with cost constraints, context-sensitive learning, feature selection for machine learning, and bias shift for learning algorithms. He received a BA, MA, and PhD in philosophy, all from the University of Toronto. He is an editorial board member for the *Journal of Artificial Intelligence Research* and belongs to the Canadian Society for Computational Studies of Intelligence and the AAAI. Contact him at the Inst. for Information Technology, Nat'l Research Council, M-50 Montreal Rd., Ottawa, ON, K1A 0R6, Canada; peter@ai.iit.nrc.ca