Automatically Exploring Hypotheses about Fault Prediction: a Comparative Study of Inductive Logic Programming Methods

William W. Cohen
AT&T Labs–Research
Florham Park, NJ 07932
wcohen@research.att.com

Premkumar T. Devanbu
Dept. of Computer Science,
University of California, Davis
devanbu@cs.ucdavis.edu

January 26, 1999

Abstract

We evaluate a class of learning algorithms known as inductive logic programming (ILP) methods on the task of predicting fault occurrence in C++ classes. Using these methods, a large space of possible hypotheses is searched in an automated fashion; further, the hypotheses are based directly on an abstract logical representation of the software, rather than on manually proposed numerical metrics that predict fault density. We compare two ILP systems, FOIL and FLIPPER, and conclude that FLIPPER generally outperforms FOIL on this problem. We analyze the reasons for the differing performance of these two systems, and based on the analysis, propose two extensions to FLIPPER: a user-directed bias towards easy-to-evaluate clauses, and an extension that allows FLIPPER to learn "counting clauses". Counting clauses augment logic programs with a variation of the "number restrictions" used in description logics, and significantly improve performance on this problem when prior knowledge is used. We also evaluate the use of ILP techniques for automatic generation of boolean indicators and numeric metrics from the calling tree representation.

1 Introduction

In this paper, we will investigate the utility of inductive logic programming (ILP) methods for the problem of classifying programs. In particular, we will explore the problem of predicting fault density in C++ classes, a problem that has received much attention in the software engineering community [Basili *et al.*, 1996; Chidamber and Kemerer, 1994]. In this problem, each training example is a C++ class definition, labeled as "positive" or "negative" according to whether faults (*i.e.*, errors) were discovered in its implementation.

The goal of most previous research in this area has been to conjecture and then test hypotheses about which properties of software reliably predict faults; reliable predictors of fault occurrence are valuable during software development, since they can be used to improve resource allocation and process management.

The process of empirically *testing* hypotheses is fairly well understood. Given a specific a hypothesis, such as *"high values of a measure α indicates fault likelihood"*, one can use a representative sample of data to test the hypothesis statistically; in this case, one might look for a statistically significant correlation between α and the occurrence of faults. The process of *generating* hypotheses is much less well understood. In previous work, hypotheses have been manually generated, based on researchers' intuitions; thus, many of the hypotheses considered in studies on fault-density prediction have been based on classical measures such as coupling and cohesion [Briand *et al.*, 1993; 1997]. Manual hypothesis generation is a well-established and valuable practice. However, advances in machine learning and hardware speed have raised another possibility.

We suggest a complementary approach, in which a large space of possible hypotheses is searched in an automated fashion, using inductive logic programming (ILP) techniques. In our approach, one first specifies a space of hypotheses; this space is defined syntactically, in terms of predicates that can derived from the source code. Next, a sample of representative data is given to the ILP algorithm, which attempts to build find a hypothesis which fits the data well. This hypothesis is then tested statistically in the usual way, using a second representative sample that was not seen by the ILP system; alternatively, methods such as cross-validation [Weiss and Kulkowski, 1990] can be used to estimate the accuracy of the hypothesis.

This approach reduces the degree of human involvement in generating hypotheses. In our experiments, no set of numerical measures is manually generated; instead, the ILP system is provided with a representation of the source code from which such measures could be derived. In particular, we use a representation of coupling relationships among C++ classes (see Table 2.1) which can be used to derive the measures of Briand *et al.* [1997] and others. The learning system then searches for logical conditions, expressed in terms of this representation, that reliably predict fault likelihood. If desired, constraints can be imposed on the sorts of logical conditions considered by the ILP system.

For several reasons, ILP systems are more suitable for this sort of task than traditional propositional learning systems. First, ILP systems do not require that examples are expressed as vectors of numbers; therefore it is possible to apply them directly to, say, a calling-tree representation of a function.

Second, ILP systems typically allow greater deal of control over the space of hypotheses

searched by the learning system than propositional systems [Adé *et al.*, 1995; Cohen, 1994a; Kietz and Wrobel, 1992]. In fault prediction, training examples are difficult to obtain, and there are many prior intuitions about domain that can plausibly be used to restrict the hypothesis space. Appropriately restricting the hypothesis space has led to improved generalization accuracy in other, similar problems, in which examples are scarce and some prior information is available [Cohen, 1994a; Pazzani and Kibler, 1992].

Third, while models that simply predict the likelihood of a fault are useful (as they can be used to focus testing) simple, comprehensible hypotheses are especially desirable, since such models could potentially be used prescriptively to avoid faulty software. In our case, the properties that we are using to predict software faults are all available at design time, and the models generated by ILP systems are compact logic programs that involve these properties. It would not be unreasonable to use broadly-validated, simple, and comprehensible hypotheses prescriptively to modify the design of the system to reduce the likelihood of faults.

Fourth, while ILP systems tend to be less efficient than propositional learning systems, this is not a substantial disadvantage in this setting, since the cost of applying even a very expensive learning method would still be dominated by the cost of data collection.

It is important to note that while ILP methods are well suited for this problem, the problem is quite different along certain technical dimensions than previously investigated benchmark problems for ILP. In particular, the data we are using contains many predicates that are highly "non-determinate" when used in the modes suggested by domain experts.[1] As we will see, these differences cause problems for some standard approaches to ILP, and also suggest some interesting extensions.

After presenting some background material, we begin by comparing the behavior of two off-the-shelf ILP systems: FOIL [Quinlan, 1990; Quinlan and Cameron-Jones, 1993] and FLIPPER [Cohen, 1995b]. We observe that FLIPPER is generally faster and achieves better results; also, surprisely, both systems produce *less* accurate theories when the hypothesis space is restricted according to expert intuitions about the problem.

Based on these experiments, we next explore the reasons for the performance differences between FOIL and FLIPPER. We propose and test a conjecture that the differences are partly due to a low-level FOIL performance optimization, and then propose an alternative strategy for improving performance on non-determinate problems. We also consider applying ILP methods in an indirect way—by first propositionalizing the data, and then using propositional methods on the converted dataset. Although this indirect method is suggested by formal results, it has been used to date on in only a small number of systems. We show that this method is useful even for this highly non-determinate problem.

Finally we consider extending the underlying hypothesis language with a variation of the *number restrictions* that are allowed in description logics, another class of learnable first-order logics [Cohen and Hirsh, 1994b; Frazier and Pitt, 1994]. This extension allows the degree of non-determinacy of a clause to be used to discriminate between positive and negative examples. It improves performance significantly in certain settings, most notably when the hypothesis space is restricted.

[1] Informally, a clause is "determinate" if every clause can be evaluated in a left-to-right order without setting any backtracking points. A predicate is determinate (with respect to an input/output mode) if clauses containing that predicate (used in that mode) will always be determinate.

2 Background

2.1 The Prediction Problem

Predicate & Modes	Description
frd(+C1,+C2)	(Friendship) Class C1 is a "friend" of class C2.
isa(+C1,+C2)	(Inheritance) Class C1 is derived from C2.
cmp(+C,?M,−P)	(Class member protection) Member function M of class C has protection P, where P∈ { `public, private, protected, extern` }.
cap(+C,?A,−P)	(Class attribute protection) Member attribute A of class C has protection P.
cmt(−C,−M,+T) cmt(+C,−M,−T)	(Class member type) Member function M of class C uses type T. The type T may be a primitive type (*e.g.*, integer) or another class. The member function "uses" type T if it takes an argument of that type, returns that type, or contains a variable of that type.
mc(+C1,−M1,−C2,−M2) mc(−C1,−M1,+C2,−M2)	(Member calls member) Member function M1 of class C1 calls member function M2 of class C2.
mp(+C1,−M1,−C2,−M2) mp(−C1,−M1,+C2,−M2)	(Member is passed member) Member function M2 of class C2 if passed as an argument to member function M1 of class C1.
arf(−C1,?C2,−A2)	(Attribute reference) Attribute A2 in class C2 has type C1, and C1≠C2.
mrf(+C1,−M1,?C2,−A2)	(Member reference) Attribute A2 in class C2 is used by member function M1 of class C1, and C1≠C2.

Table 1: Relations available for training

The data used in this paper was collected in an empirical study conducted at the University of Maryland [Basili *et al.*, 1996]. In the study, students were required to develop a medium-sized information management system. The final projects were then tested by an independent group of software professionals, and (among other things) the components that contained faults were recorded.

The implementations were written in C++, a widely-used object-oriented extension of C. Software in C++ is organized into *classes*, each of which implements a different abstract

datatype. Classes are organized into an inheritance hierarchy. The attributes (*i.e.*, data fields) and functions associated with a class are called *members*. The members of a class can be either *private*, *public*, or *protected*. The private members of class C are accessible only from other members of C. Protected members are accessible only from members of C or its descendants. Public members can be accessed from anywhere. Additionally another class C' can be declared to be the *friend* of C, which allows the members of C' to access the private members of C.

In the study, faults were assigned to classes; thus every class became an example in our dataset. Classes were marked positive if they had any faults, and negative otherwise. There were 122 classes, 58 of which had faults. The development and validation of metrics for fault prediction is an active area of investigation in the software engineering community [Basili *et al.*, 1996; Fenton, 1992; Chidamber and Kemerer, 1994] Recently a set of metrics have been proposed [Briand *et al.*, 1997] that are based on "coupling" between classes; informally, class C_1 and C_2 are "coupled" if C_1 uses C_2 in some way, or if C_2 uses C_1. The computation of these metrics is based on a set of relations that encode various types of coupling relationships between classes, as well as certain other possibly useful information, such as the inheritance hierarchy. These relations, described in Table 1, were extracted from the source code using the program analysis tool GEN++ [Devanbu and Eaves, 1994]. In all, 20,929 ground facts concerning these relations and the 122 examples were extracted; henceforth these will be called the *background facts*.

The background facts describe various sorts of inter-class coupling relationships, and can be used to derive values for coupling metrics. However, they also may potentially support hypotheses unrelated to coupling. For instance, the rule "faulty(C) $\leftarrow$ mc(C,M,C,M)" predicts that a class C is faulty if it has a member function M that calls itself recursively. ILP algorithms produce hypotheses of this sort; each hypothesis is a set of Prolog-like rules, using the relations defined by the background facts and the fault data. The possible set of such rules is enormous, and testing all of them is infeasible, so the ILP systems constrain the search for hypotheses in two ways. First, they search for a single good hypothesis in a greedy manner; secondly, they allow search to be constrained by syntactic restrictions on the types of clauses that appear in a hypothesis.

One common way that ILP systems restrict the space of possible clauses is by *mode restrictions*. Every Prolog rule (more properly called a trmclause) consists of a trmhead or conclusion, and *body*, which is a conjunction of goals. Thus, in a clause such as

p(X,Y,Z) $\leftarrow$ q(X,Y), r(Y,Z)

"p(X,Y,Z)" is the head, and "q(X,Y)" and "r(Y,Z)" are *literals* in the body. Under the assumptions made by the ILP systems used in this paper, every Prolog clause will be invoked with all variables in the head bound to constant values, and each variable X in the body of the clause will be bound at the first literal L in which it appears (reading left to right). We say that X is an *output variable* of the literal L if it is bound by L, and an *input otherwise*. In the list of suggested modes, "+" indicates a position must contain an input variable, "−" indicates a position must contain an output, and "?" indicates either an input or output position.

Table 1 gives for each coupling relation a set of suggested input/output modes. If predicates are used in the suggested modes, then the resulting theories are more likely to contain coupling relationships, some of which are (by hypothesis) meaningful predictors of faults. Adopting these modes thus is one means of injecting prior knowledge into the learning process.

2.2 ILP Systems

Our first experiments were carried out with two off-the-shelf ILP systems, FOIL6.4 [Quinlan, 1990; Quinlan and Cameron-Jones, 1993] and FLIPPER [Cohen, 1995b]. Later experiments were carried out with modified versions of these systems. We will now briefly describe FOIL and FLIPPER.

2.2.1 Background on ILP

The ILP systems used in this paper learn logic programs in certain constrained forms. The input to an system consists of a set of predefined *background predicates*, such as the predicates `frd`, `mc`, ... described in the previous section, and a series of labeled positive and negative examples of the form

$$\pm p(t_{1,1}, \ldots, t_{1,n_1}), \pm p(t_{2,1}, \ldots, t_{2,n_2}), \ldots$$

where each $t_{i,j}$ is a ground term. In the fault prediction problem, for instance, the examples are of the form $+$`faulty(t)` or $-$`faulty(t)`, where `t` is a constant naming a C++ class. The logic program learned by the ILP system is a set of clauses of the form

$$\begin{aligned} &p(X_1, \ldots, X_n) \leftarrow body_1 \\ &p(X_1, \ldots, X_n) \leftarrow body_2 \\ &\vdots \\ &p(X_1, \ldots, X_n) \leftarrow body_k \end{aligned}$$

where each $body_i$ is a conjunction of literals; in this paper, these conjunctions refer only to the background predicates, and are non-recursive. A new instance $p(u_1, \ldots, u_n)$ will be classified as positive by the learned program if it is proved true by the program, and classified as negative otherwise.

In the ILP systems used in this paper, the literals in the body are always of the form of the form $q(Y_1, \ldots, Y_k)$, or $\neg q(Y_1, \ldots, Y_k)$ where q is a *background predicate*. Negative literals in the $body_i$'s are interpreted using the Prolog rule of negation as failure, and each background predicate is defined by a set of ground facts (such as `frd(widgetClass,windowClass)`).

This learning problem is a generalization to relational concepts of the learning problem addressed by attribute-value learning systems that learn concepts in disjunctive normal form [Michalski *et al.*, 1986; Pagallo and Haussler, 1990; Cohen, 1995a].

2.2.2 The FOIL learning algorithm

FOIL learns function-free Prolog predicate definitions from examples and background knowledge. Both examples and background knowledge are encoded as ground facts. Since FOIL's

function FOIL(Data):
begin
    PredDef := an empty predicate
    while Data contains positive examples **do**
        Clause := GrowClause(Data)
        add Clause to PredDef
        remove examples covered by
          Clause from Data
    endwhile
    return PredDef
end

function GrowClause(Data):
begin
    Clause := a clause with an empty body and
      maximally general head: $p(X_1, \ldots, X_k) \leftarrow$true
    while Clause covers some negative examples **do**
        Body := the body of Clause
        Head := the head of Clause
        replace Clause with the new clause Head$\leftarrow$ Body,NewLit
          that maximizes Information-Gain(Clause,NewClause,Data)
          where NewLit is some new condition
    endwhile
    return Clause
end

function Information-Gain(Clause$_i$,Clause$_{i+1}$,Data):
begin
    return $T^{++} \times \quad -\log_2 \frac{T_i^+}{T_i^+ + T_i^-} + \log_2 \frac{T_{i+1}^+}{T_{i+1}^+ + T_{i+1}^-}$ ¶

    where T_j^+ (respectively T_j^-) is the number of positive
      (respectively negative) examples covered by Clause$_j$, and
      T^{++} is the number of positive examples covered by Clause$_{i+1}$
end

Figure 1: The basic FOIL algorithm

```
function IREP*(Data)
begin
  Data0 := copy(Data);
  PredDef := an empty predicate
  while Data contains positive examples do
    /* grow and prune a new clause */
    split Data into GrowData,PruneData
    Clause := GrowClause(GrowData)
    Clause := PruneClause(Clause,PruneData)
    add Clause to PredDef
    remove examples covered by
      Clause from Data
    /* check stopping condition */
    if DL(PredDef)>DL(PredDef_opt) + d
      where PredDef_opt has lowest DL
      of any PredDef constructed so far
    then
      break out of the while loop
    endif
  endwhile
  PredDef := Compress(PredDef,Data0)
  return PredDef
end
```

```
function Optimize(PredDef,Data)
begin
  for each clause c ∈ PredDef do
    split Data into GrowData,PruneData
    c' := GrowClause(GrowData)
    c' := PruneClause(c',Prunedata)
      guided by error of PredDef-c+c'
    ĉ := GrowClauseFrom(c,GrowData)
    ĉ := PruneClause(ĉ,Prunedata);
      guided by error of PredDef-c+ĉ
    replace c in PredDef with best of c, c', ĉ
      guided by DL(Compress(PredDef-c+x))
  endfor
  return PredDef
end
```

```
function FLIPPER(Data)
begin
  PredDef := Optimize(IREP*(Data),Data)
  UncovData := examples in Data not
    covered by clauses in PredDef
  return PredDef + IREP*(UncovData)
end
```

Figure 2: The FLIPPER algorithm

learning algorithm is described elsewhere we will only summarize it briefly. The basic FOIL algorithm is shown in Figure 1.

FOIL is a set-covering type algorithm—*i.e.*, it starts with an empty predicate definition, and then extends the definition by adding one clause at a time. After each clause is constructed, the examples covered by that clause are deleted, and this process continues until no more positive examples remain.

Clauses are constructed using a greedy method. FOIL begins with a clause with an empty body, and repeatly specializes the clause body by adding literals, guided by an information-theoretic measure of clause quality called "information gain". FOIL considers adding any single literal `NewLit` to the body of a clause, as long as it satisfies the mode and typing constraints given by the user, and as long as it shares at least one variable with the existing clause. Algorithms like FOIL that combine set covering and greedy construction of clauses are often called *separate and conquer* algorithms.

FOIL6.4 adds a number of refinements to the basic algorithm outlined above, including an method for handling noisy data by stopping construction of a clause (or predicate definition) before it is completely consistent with the data. This stopping criterion is guided by a minimum description length (MDL) heuristic. Some sort of noise immunity is important on a problem like fault density estimation, in which it is implausible to expect that concept membership will be completely determined by the available background relations. A more complete description of FOIL can be found elsewhere [Quinlan, 1990; Quinlan and Cameron-Jones, 1993].

2.2.3 The FLIPPER learning algorithm

Like FOIL, FLIPPER learns function-free Prolog predicate definitions from examples and ground background relations. Algorithmically, however, FLIPPER is quite different; it is a first-order version of RIPPER [Cohen, 1995a], which in turn is based on the *incremental reduced error pruning* (IREP) algorithm [Fürnkranz and Widmer, 1994].

The first stage of FLIPPER is a variant of IREP that we call IREP∗. (See Figure 2.) IREP∗ is a set-covering algorithm: it constructs one clause of a predicate definition at a time, and removes the examples covered by a new clause c as soon as c is added to the predicate definition. To build a clause, IREP∗ uses the following strategy. First, the uncovered examples are randomly partitioned into two subsets, a *growing set* containing 2/3 of the examples and a *pruning set* containing the remaining 1/3. Next, a clause is "grown" by repeatedly adding literals to an empty clause guided by information gain on the growing set. Addition of new literals (growing) continues until the clause covers no negative examples in the growing set. After growing a clause, the clause is immediately *pruned* (*i.e.*, simplified). Our implementation considers deleting any final sequence of conditions from the clause, and chooses the deletion that maximizes the function $\frac{p-n}{p+n}$ where p (respectively n) is the number of positive (negative) examples in the pruning set covered by the new clause. After pruning, the pruned clause is added to the predicate definition, and the examples covered by it are removed.

After each clause is added, the total *description length* of the current predicate definition and the examples is computed.[2] FLIPPER stops adding clauses when this description length is more than d bits larger than the smallest description length obtained so far, or when there are no more positive examples.[3] The predicate definition is then *compressed* by examining each clause in turn, starting with the last clause added, and deleting clauses so as to minimize description length.

This greedy process can lead to a suboptimal predicate definition, thus after FLIPPER stops adding clauses, the predicate definition is "optimized". Clauses are considered in turn, in the order in which they were added, and for each clause c_i, two alternative clauses are constructed. The *replacement for* c_i is formed by growing and then pruning a clause c_i', where pruning is guided so as to minimize error of the entire predicate definition (with c_i' replacing c_i) on the pruning data. The *revision of* c_i is formed analogously, except that it is grown by greedily adding literals to c_i, instead of to an empty clause. Finally a decision is made as to whether the final theory should include the revised clause, the replacement clause, or the original clause. This decision is made using the description length heuristic—the definition with the smallest description length after compression is preferred. After optimization, the definition may cover fewer positive examples; thus IREP∗ is called again on the uncovered positive examples, and any additional clauses that it generates are added.

Both FLIPPER and FOIL6.4 are both relatively complex learning algorithms, and both are direct descendents of the basic FOIL algorithm presented in Figure 1. The experi-

[2]The description length of a clause is defined to be $\sum_i \log_2 n_i$ where for n_i is the number of possible refinements that were considered in the i-th stage of constructing the clause, and the scheme for encoding errors is the same as that used in the latest version of C4.5rules [Quinlan, 1995].

[3]By default $d = 64$.

mental support for the particular extensions embodied in FLIPPER is described elsewhere [Fürnkranz and Widmer, 1994; Cohen, 1995a; 1995b]. FLIPPER, like FOIL, is implemented in C, and uses indexed joins internally to speed up evaluation of the function-free Prolog clauses that it hypothesizes. FLIPPER differs from FOIL in many other ways, most of which are not important for this problem. One important difference is that unlike FOIL, FLIPPER has a "declarative bias"—*i.e.*, the user can define the space of clauses searched by FLIPPER by writing expressions in a declarative language [Cohen, 1995b]. (In this paper, we will use the term "bias" to refer to the hypothesis space searched by a learning program. A small hypothesis space corresponds to a "strong bias" and a large hypothesis space corresponds to a "weak bias".)

To clarify this difference, note that the clauses constructed by FLIPPER and FOIL, as described above, that are built by repeatedly adding a new literal to the end of a clause body. In FOIL, almost any literal can be added; thus FOIL explores a very wide range of possible clauses. The user can control the set of clauses generated using a few specific techniques, notably by restricting the mode of predicates, and by command-line arguments that suppress generation of negative literals or recursive literals.

In contrast, FLIPPER employs a declaratively specified *refinement operator* [Shapiro, 1982], which allows the user to specify exactly how a clause can be extended. (For details of the refinement language, see [Cohen, 1995b].) The refinement operator gives the user fine control over the space of hypotheses that is searched by the learning system. However, in most of the experiments described below, this feature of FLIPPER was not used; instead we wrote a script that converts a set of typed mode declarations, such as are accepted as input by FOIL, to a FLIPPER refinement language definition for the corresponding search space.

3 Experimental Results with FOIL and FLIPPER

3.1 Initial experiments

Employing the experimental methodology used by Srinivasan *et al.* [1996] in a similar small-sample situation, we estimated error rates with 10-fold cross-validation. The examples were randomly divided into 10 disjoint "folds" of approximately equal size. We then trained each learner on nine of the ten folds, and tested on the remaining fold.

Following this methodology, all 122 examples are used (at some point) as test cases. The predictions made by each learner on the test cases were recorded, and McNemar's test was used to compare accuracy of different learners. To compare learners L_1 and L_2 with this test, one determines if the probability $P(L_1 \text{ is correct} \mid L_1 \text{ and } L_2 \text{ disagree})$ is significantly different from 0.5, as estimated by a 2-tailed test with the binomial distribution. If it is, then one can with high confidence reject the null hypothesis that L_1 and L_2 perform comparably on the test cases.

As a special case, one can use McNemar's test to compare a learner's performance with the performance of the default hypothesis. (On this problem, the default hypothesis always predicts a class to be fault-free). This comparison indicates if a learner's hypothesis is significantly better than just guessing the most likely class. This methodology is discussed and evaluated more fully in Appendix B.

	Learning system (options)	Errors #	%	*vs.* default W-L	p	Time (sec) avg	max
	FOIL6.4	55	30.9	18-15	0.51	6,838	10,672
	(monotone)	50	28.1	20-12	0.86	11,540	16,636
	(monotone,modes)	53	29.8	8-3	0.89	35,451	63,185
$\surd$F1	FLIPPER	42	23.6	28-12	0.99	495	1,317
$\surd$F2	(monotone)	40	22.5	29-11	0.99	1,773	7,714
	(monotone,modes)	49	27.5	22-12	0.87	3,944	24,439
F3	(strong bias)	62	34.8	25-29	0.41	18	23

Table 2: Comparison of off-the-shelf learning systems on fault prediction

Table 2 summarizes our first results on the fault density problem. We give for each learning system the error rate as estimated by the cross-validation, both as the number of errors and as a percentage; the result of comparing the classifier with the default classifier using McNemar's test; and the average and maximum run time of the ten learning trials.[4] For McNemar's test, we give the won-loss record of the learning system versus the default classifier on those test cases for which they disagree, and the confidence p with which one can reject the null hypothesis that the learner's performance is mere random deviation from the default hypothesis.

We began by comparing FOIL with its default settings to FLIPPER with the nearest possible approximation to this bias. Since we wished to explore the effect of prior knowledge on learning, we also ran FOIL and FLIPPER with some more restrictive biases, making use command-line arguments for FOIL, and FLIPPER's refinement language. First, we suppressed use of "negative" literals, of the form ¬L; the rationale for this is that the coupling relationships describable by the background predicates should make faults more likely, rather than less likely. These results appear in table as "monotone" FOIL and FLIPPER. Secondly, we ran the monotone versions of FOIL and FLIPPER with the mode restrictions given in Table 1.

With the standard settings, and with the monotone bias, FLIPPER's average runtime was about 5-10 times faster than FOIL's. FLIPPER also gives better results with respect to error rate; for these biases, FLIPPER's results are statistically significantly better than FOIL's.[5] Further, only FLIPPER performs statistically significantly better than the default classifier (indicated with a checkmark $\surd$ in the table).

The monotone bias appears to decrease the error rate somewhat for both systems, although the decrease is not significant in either case. However, injecting knowledge in the form of mode restrictions gives disappointing results, raising the error rate for both FOIL and FLIPPER (although not significantly). Somewhat surprisingly, the run time for both FOIL and FLIPPER is also dramatically worse with the more restricted bias induced by the

[4] On a 250MHz MIPS Irix computer.

[5] Using McNemar's test, FLIPPER is better with probability $p > 0.98$ on a two-tailed test in the monotone case and $p > 0.988$ in the non-monotone case.

mode declarations, with both learners occasionally suffering enormously long run-times.

Line F3 represents another attempt to inject prior knowledge into the learning process. It indicates FLIPPER's performance given a highly restricted bias suggested by the metrics proposed by Briand *et al.* [1997]; this hand-coded bias also increases the error rate. These restrictions allow only 110 distinct clauses, each consisting of one literal with one of the predicates `mp`, `mc`, `cmt`, `arf`, or `mrf`, used on one of the suggested modes; followed optionally by either a single `frd` or `isa` literal, possibly negated, or a conjunction of negated `frd` and `isa` literals that collectively ensure that the two classes mentioned in the first literal are not related in any way by either inheritance (`isa`) or friendship (`frd`).

The increase in error rate with prior knowledge is disappointing for several reasons—not the least of which is that generalizing and validating the Briand *et al.* metrics was the original motivation for this work. More generally, one usually prefers hypotheses that are comprehensible to domain experts, since this increases one's confidence in the correctness and robustness of these hypotheses. This is especially true in a situation like this one, in which little data is available to validate a hypothesis.

In the remainder of the paper we will investigate the reasons why this apparently plausible prior knowledge degrades performance, and propose some extensions to FLIPPER which enable it to make more effective use of the prior knowledge. We will also investigate further the reasons for the differences in run-time and generalization error between FLIPPER and FOIL.

3.2 Runtime Differences

We conjecture that the increase in runtime when modes are used is in part a consequence of the highly non-determinate nature of this dataset. Recall that a predicate is *determinate* for a particular mode if there will always be at most one way of binding the output variables given the input variables. For example, if the predicate `child(X,Y)` is true when `Y` is a child of node `X`, then `child` will be determinate for the mode `child(-,+)` in a tree data structure, but non-determinate for the mode `child(+,-)`. Determinate predicates are especially easy to evaluate, since no backtracking points need to be set, and in ILP systems, determinate predicates are often treated specially; for example, FOIL6.4 has special mechanisms for handling determinate predicates, and GOLEM [Muggleton and Feng, 1992] allows only determinate predicates. Formal results also suggest that clauses containing determinate predicates are especially easy to learn [Džeroski *et al.*, 1992; Cohen, 1995c].

Unfortunately, many of the predicates used in this problem are highly non-determinate; in particular, the predicates `mc`, `mp`, `mrf`, and `arf` can be instantiated many different ways for some examples, when used in their suggested modes. For instance, given a binding for `C1`, there are many ways the literal `mc(C1,M1,C2,M2)` can be instantiated—in fact, almost every function call in a method of `C1` leads to a distinct binding for `M1`, `C2`, and `M2`. A long clause containing many of these non-determinate predicates can be extremely expensive to evaluate.

While the mode declarations reduce the search space, they do not exclude these expensive clauses. Thus if the learning system is unlucky enough to construct expensive clauses while searching for a hypothesis, learning will be slow. It appears that on this dataset, expensive

clauses are more likely to be constructed when mode restrictions are used. Presumably this is because absent the mode restrictions, there are some compact, inexpensive clauses that are highly predictive.

To test this conjecture, we extended FLIPPER to accept an additional parameter M which limits the number of distinct proofs that any individual example may have. Here a *proof for an example* is defined to be a binding for the variables of the clause body that makes the clause body true, assuming that the head of the clause is unified with the example. When M is given, then any candidate clause that can be proved to be true more than M distinct ways for any uncovered example is discarded. In other words, a clause will be discarded if, given bindings for variables in the head, it can be made true by using more than M distinct combinations of assignments to the variables in the body of the clause. Specializations of such a clause will probably require a large amount of backtracking to evaluate, and hence will be expensive to evaluate. By appropriately restricting M, a user can avoid expensive clauses, without having to commit to any particular syntactic restriction on clauses such as determinacy [Muggleton and Feng, 1992] or locality [Cohen, 1994b].

	Learning system (options)	Errors #	Errors %	*vs.* default W-L	*vs.* default p	Time (sec) avg	Time (sec) max
$\sqrt{}$F1	FLIPPER	42	23.6	28-12	0.99	495	1,317
$\sqrt{}$F2	(monotone)	40	22.5	29-11	0.99	1,773	7,714
	(monotone,modes)	49	27.5	22-12	0.87	3,944	24,439
M1	(M=100,monotone,modes)	63	35.4	21-26	0.46	384	2,454
M2	(M=500,monotone,modes)	52	29.2	31-25	0.57	989	3,470

Table 3: Effect of limiting non-determinacy

Lines M1-M2 of Table 3 show the performance of FLIPPER with modes and with M set to 100 and 500. Even $M = 500$ gives a dramatic improvement in run-time, supporting the conjecture. This was true even though implementing this extension required us to disable one optimization made by FLIPPER.[6]

Unfortunately, the restriction also adversely affects accuracy. The increase in error for $M = 100$ versus no restriction is statistically significant ($p > 0.95$), leading to the surprising conclusion that it is helpful to include these extremely non-deterministic clauses in the search space.

3.3 Error Rate Differences

As a further test, we constructed an artificial dataset that is similar to our fault density dataset, except that the degree of non-determinacy and the amount of noise can be varied.

[6]Specifically, FLIPPER's inner loop evaluates a clause $A \leftarrow B_1, \ldots, B_k, L$ on a set of examples S. This clause is always a refinement of a previous clause $A \leftarrow B_1, \ldots, B_k$ which is known to cover all the examples in S. Thus FLIPPER normally performs a dependency analysis to determine which of the literals $B_1, \ldots, B_k$ are necessary in binding the variables in L, and then evaluates a simplified clause in which unnecessary literals are deleted. Unfortunately deleting such literals changes the number of distinct proofs of a clause.

	Learning system (options)	Errors #	Errors %	*vs.* default W-L	*vs.* default p	Time (sec) avg	Time (sec) max
	FOIL6.4	55	30.9	18-15	0.51	6,838	10,672
$\sqrt{}$F1	FLIPPER	42	23.6	28-12	0.99	495	1,317
P1	FLIPPER/gain on proofs	58	32.6	0-0	0.00	91	100

Table 4: Comparison of learning systems on fault prediction

We now turn to the differences in error rate between FLIPPER and FOIL. FLIPPER has been previously shown to obtain lower error rates than FOIL on some learning problems with a high level of noise, but little non-determinacy [Cohen, 1995b]; thus one likely explanation for the difference in generalization performance is simply the differences in pruning strategies used by the two systems. However, the non-determinacy of the dataset suggests an additional explanation for why FLIPPER's generalization error is less than FOIL's.

For efficiency reasons, when computing information gain, FOIL does not compute the number of positive and negative examples covered by a candidate clause, as shown in Figure 1. Instead, FOIL counts the number of distinct *proofs* of all positive (respectively negative) examples.[7] In highly non-determinate domains, these counts can be quite different from the number of examples covered by a clause—and intuitively it is number of examples, not proofs, that is important in learning. Thus one would expect some degradation in performance due to counting proofs, rather than examples. Further, one would expect this effect to be most serious domains that are both highly indeterminate and noisy; when domains are indeterminate, then proof counts and example counts will differ most; and when domains are noisy, it is most difficult to choose the best clause refinement.

To test this, we modified FLIPPER so that information gain is computed *á la* FOIL—from the number of proofs, rather than the number of examples. We conjectured that this would increase the generalization error. Line P1 of Table 4 shows the results for this variant of FLIPPER, using a monotone bias; it has a significantly higher error rate than standard FLIPPER ($p > 0.99$), supporting the conjecture.

	time(FOIL)-time(FLIPPER)					error(FOIL)-error(FLIPPER)				
k	β=0.0	0.1	0.2	0.25	0.3	β=0.0	0.1	0.2	0.25	0.3
1	−27.6	−244.7	−212.9	−225.6	−262.1	0.0	17.2	19.6	−3.3	40.4
2	−71.2	−680.4	−993.0	−983.2	−720.4	0.0	13.8	23.8	28.1	36.1
5	−148.7	−2327.2	729.7	>6588.9	>4465.8	0.0	5.9	14.0	***	***
10	−271.9	−4608.4	−889.2	>4922.1	>5930.5	0.0	7.9	17.3	***	***
20	−686.3	1518.5	>6463.3	>5130.2	>4439.9	0.0	9.3	***	***	***

Table 5: Comparison of FOIL and FLIPPER on artificial data

(See Appendix A for details.) We varied the maximum amount of non-determinacy from $k = 1$ to $k = 20$, where $k = 1$ corresponds to a determinate dataset, and varied noise from $\beta = 0$ to $\beta = 0.3$, where $\beta = 0$ corresponds to a noise-free dataset. We then compared FOIL and FLIPPER as these parameters were varied. Both the systems were run with a time bound of 10,000 CPU seconds, and FLIPPER was run with $M = 100$. Table 5 summarizes the results of this experiment, showing the differences in run-time (or a bound on this difference, if one system did not complete in 10,000 seconds) and also the differences in error rate (for those cases where both systems completed in 10,000 seconds).

FLIPPER nearly always achieves lower error rates when there is noise in the data. This difference is significant,[8] and appears to hold regardless of the amount of non-determinacy. With respect to run-time, FOIL is generally much faster—except when there is both a large amount of noise and a noticeable degree of non-determinacy. These, of course, are the conditions that prevail in the natural fault density dataset. FOIL seems to be slow on these problems because it often "overfits" noisy data, and constructs hypotheses containing many long clauses; in non-determinate domains these are very expensive.

We also compared the error rates of standard FLIPPER with the proof-counting variant on the same artificial datasets. Of nine non-zero differences, standard FLIPPER has a lower error rate eight times, a statistically significant difference.[9] This again supports the hypothesis that counting proofs rather than examples is inadvisable in non-determinate noisy domains.

4 Experiments with other learning methods

4.1 ILP With Number Restrictions

So far, we have considered only off-the-shelf learning methods, and some minor variants of these methods. We will now consider some more novel learning algorithms that are suggested by the analysis.

ILP methods represent hypotheses as logic programs. Another class of first-order languages are *description logics*, sometimes also called *terminological logics* or *KL-ONE-type languages* [MacGregor, 1991; Woods and Schmolze, 1992]. Several experimental systems learn description logics [Cohen and Hirsh, 1994b; Kietz and Morik, 1991], and formal results show that some fairly expressive description logics are pac-learnable [Cohen and Hirsh, 1994a; Frazier and Pitt, 1994].

One intriguing property of description logics is that they make it possible to succinctly write some concepts that are quite cumbersome to express in Prolog. For instance, given the predicate `child`, the set of people with at least three different children would be written `(AND PERSON (AT-LEAST 4 CHILD))` in the description logic CLASSIC, whereas in Prolog it would be written as the possible substitutions for X for the query

$\leftarrow$ person(X), child(X,Y1), child(X,Y2), child(X,Y3),child(X,Y4),
Y1$\neq$Y2, Y1$\neq$Y3, Y1$\neq$Y4, Y2$\neq$Y3, Y2$\neq$Y4, Y3$\neq$Y4

[8] With $p > 0.99$ on a two-tailed sign test.

[9] With $p > 0.99$ on a two-tailed sign test.

It is plausible that such number restrictions could be useful in a problem such as fault density prediction, in which there are many relations that vary widely in non-determinacy.

We thus extended FLIPPER to learn a class of logic programs with number restrictions—specifically logic programs containing what we will call *counting clauses*. A counting clause is an ordinary Prolog clause that has an associated threshold k. An example e is classified as positive by the clause only if the clause has at least k proofs—*i.e.*, if given the bindings imposed by unifying the head of the clause with e, there are at least k ways to bind the non-head variables in the clause that make the clause true.

Informally, to cover an example e, a counting clause not only must be true (given the bindings imposed by unifying with e); it must be true for at least k different reasons, where k is the associated threshold. If $k = 1$ then the clause is an ordinary non-counting clause. As an example, the following clause with threshold $k = 4$ covers an example $p(e)$ iff e has at least 4 children: p(X) ← person(X), child(X,Y).

It should be emphasized that counting clauses are number restrictions on *conjunctions* of predicates, thus allowing one to also easily express concepts like "people with at least three female children", "people with at least three children enrolled in an Ivy-league college", as well as "people with at least three children". In particular, augmenting the background knowledge with ordinary arithmetic tests and a small number of additional predicates would not provide the expressive power of counting clauses—for instance, introducing the predicate "number_of_children" allows one to succinctly express only the simplest of the examples above.

In contrast to the previous extension we described, which seeks to limit the cost of non-determinism, counting clauses are a means of *exploiting* non-determinism, by allowing the degree of non-determinism to be used in prediction. In fault density prediction, counting clauses allow a hypothesis to classify based on the *degree* of some type of coupling, rather than whether that type of coupling exists.

The changes to the algorithm required to implement this extension were minor, and affected only the procedures `GrowClause` and `GrowClauseFrom`. Whenever a new literal L is added to a clause c, the clause is tested against each example e, and the largest threshold k such that e is covered is computed. Based on this information, the threshold k that optimizes information gain for the extended clause is determined, and associated with the refined clause. In pruning the clause, these refinements are simply "undone"—in other words, when a final sequence of literals in a clause is deleted, the threshold k is restored to whatever threshold was adopted before those literals were added. We applied this extension to two biases: the monotone bias with modes, and the hand-coded bias suggested by the metrics of Briand *et al.* [Briand *et al.*, 1997]. Counting clauses increase the search space somewhat, and also require one of the optimizations made by FLIPPER to be disabled;[10] perhaps because of this, learning counting clauses can be relatively expensive. We thus limited non-determinism to $M = 100$ and $M = 500$ with the mode declarations. (The hand-coded bias is so restricted that run-time performance is not an issue.)

The results are shown in Lines C1-C3 of Table 6. There is an improvement over standard FLIPPER in each case. This improvement is statistically significant for the two more highly restricted biases—the hand-coded bias ($p > 0.999$) and mode restrictions with $M = 500$

[10]The dependence analysis optimization was disabled, for the reasons described in Section 3.2.

	Learning system (options)	Errors #	Errors %	*vs.* default W-L	*vs.* default p	Time (sec) avg	Time (sec) max
	FOIL6.4	55	30.9	18-15	0.51	6,838	10,672
	(monotone)	50	28.1	20-12	0.86	11,540	16,636
	(monotone,modes)	53	29.8	8-3	0.89	35,451	63,185
$\surd$F1	FLIPPER	42	23.6	28-12	0.99	495	1,317
$\surd$F2	(monotone)	40	22.5	29-11	0.99	1,773	7,714
	(monotone,modes)	49	27.5	22-12	0.87	3,944	24,439
F3	(strong bias)	62	34.8	25-29	0.41	18	23
M1	(M=100,monotone,modes)	63	35.4	21-26	0.46	384	2,454
M2	(M=500,monotone,modes)	52	29.2	31-25	0.57	989	3,470
	FLIPPER/counting clauses						
C1	(M=100,monotone,modes)	48	27.0	32-22	0.82	1,386	4,369
$\surd$C2	(M=500,monotone,modes)	41	23.0	37-20	0.97	4,737	9,917
$\surd$C3	(strong bias)	39	21.9	34-15	0.99	29	42
$\surd$C4	(biased,optimize 2x)	35	19.7	39-16	0.99	28	32

Table 6: Comparison of learning systems on fault prediction

($p > 0.95$).

One final experiment was performed using FLIPPER. As noted in Section 2.2, FLIPPER "optimizes" its clauses after it learns them. In RIPPER, the propositional version of FLIPPER, this optimization step is repeated twice by default; the second iteration tends to give a slight additional improvement, and is omitted in FLIPPER only because of the greater expense of first-order learning. Since learning with the hand-coded bias is relatively inexpensive, however, we tried FLIPPER with counting clauses and two rounds of optimization. As shown in line C4, this offers an additional improvement in error rate, giving the lowest error rate we have achieved this problem.

4.2 Indirect ILP methods

Another experiment was motivated by the observation that the hand-coded bias generates a small finite set of clauses. This bias can be encoded propositionally by introducing one attribute a_i for each possible clause c_i. We considered two variants of this encoding. In the first, a_i is boolean, indicating whether c_i covers an example e. In the second, a_i is numeric, indicating the number of distinct proofs that c_i covers e. We then ran two propositional learners, C4.5 [Quinlan, 1994] and RIPPER [Cohen, 1995a], on the constructed dataset. The results are again shown in Table 2; lines I1-I2 summarize the results for the boolean encoding, and lines J1-J2 summarize the results for the numeric encoding.

In both cases, the numeric encoding improves performance over the boolean encoding—significantly for RIPPER ($p > 0.99$).[11] This result can be viewed as a propositional analog

[11]The differences between C4.5 and RIPPER on the same datasets are not statistically significant.

	Learner	Options	k	Errors #	Errors %	*vs.* default W-L	*vs.* default p	# Features
I1	RIPPER	biased		59	33.1	20-21	0.20	111b
I2	C4.5	biased		52	29.2	25-19	0.64	111b
I3	RIPPER	monotone	1	47	26.4	25-14	0.93	35b
$\surd$ I4	RIPPER	monotone	2	42	23.6	37-21	0.96	4100b
I5	RIPPER	monotone,modes	1	59	33.2	19-20	0.11	19b
I6	RIPPER	monotone,modes	2	50	28.1	35-25	0.70	490b
$\surd$ J1	RIPPER	biased		37	20.8	40-19	0.99	111n
J2	C4.5	biased		44	24.7	37-23	0.93	111n
J3	RIPPER	monotone	1	45	25.3	37-24	0.91	35n
$\surd$ J4	RIPPER	monotone	2	36	20.2	41-19	0.99	4100n
$\surd$ J5	RIPPER	monotone,modes	1	37	20.8	40-19	0.99	19n
$\surd$ J6	RIPPER	monotone,modes	2	44	24.7	32-18	0.95	490n

Table 7: Results with indirect ILP

to the improvement gained by extending FLIPPER with counting clauses.

These experiments also suggest using an alternative type of ILP. Both FOIL and FLIPPER learn directly from examples and background facts. An alternative approach to ILP is to first generate logic program clauses, and then use these clauses convert the examples and background knowledge into a propositional form. One can then apply conventional propositional learning methods to the constructed dataset. A number of theoretically proposed learning methods [Lavrač and Džeroski, 1992; Džeroski *et al.*, 1992; Cohen, 1994b] and at least one practical learning system (LINUS [Lavrač and Džeroski, 1994]) rely on this technique.

We evaluated the following simple indirect learning system. Given a dataset, a set of background relations, and a set of suggested modes, we enumerated all possible clauses with at most k body literals. Each of these clause c corresponds to a boolean feature a_c, which is true for an example x iff c covers x.

The approach is suggested by formal results concerning the language of *k-local clauses*. Define X to *touche* Y if X and Y are two variables appearing in a clause, and define *influences* to be the transitive closure of *touches*. A *k-local* clause is one in which every variable appearing in the body but not the head of the clause influences variables appearing in at most k different literals.

It has been shown that the language of k-local clauses is the most expressive that can be efficiently learned in certain formal models [Cohen, 1994b]. Any monotone DNF formulae over the constructed features can be converted to a k-local Prolog predicate definition. Hence applying a learning system like RIPPER to the propositionalized ruleset is a plausible approach to learning k-local clauses.

The results of using RIPPER in this way (for some of the more promising biases) are shown in lines I3-I6 of Table 7. With $k = 1$, the indirect approach is significantly worse

($p > 0.99$) than the comparable direct algorithm, monotone FLIPPER (lines I3,I5); however it is statistically indistinguishable from FLIPPER with $k = 2$ (lines I4,I6). Like the direct approach, the indirect approach fairs poorly with restricted biases (lines I5-I6).

More interestingly, a corresponding set of numeric features can be constructed using the same method used in constructing lines J1-J2 of the table; specifically, for each length-k clause c and each example e one can construct a numeric feature indicating the number of distinct proofs that c covers e. To our knowledge, this approach has not been used before real-world problems. RIPPER performs rather well using these feature sets—in all cases but one significantly outperforming the default classifier (lines J3-J6).

In general, learning time on the converted datasets is much faster than with the indirect methods, ranging from well under a second for the smaller feature sets to around 30 seconds for the $k = 2$ bias. However, the cost of propositionalizing can be high.[12]

5 Summary of all experiments

As a final summary, Table 8 lists all the experimental results obtained on this fault prediction problem. All the labels are as given above in the text.

Notice that given appropriate features, the propositional systems perform well on this problem—in particular, their error rates are not statistically significantly worse than even the best of the ILP systems.[13] Similar results have been found elsewhere [Srinivasan *et al.*, 1996].

The benefit of general-purpose ILP methods is illustrated by lines F1-F2 and C2-C4, in which very good generalization performance is obtained, without any numerical metrics being explicitly defined. In spite of the larger search space considered by the ILP methods, these results are not significantly worse than the best results obtained using numeric metrics designed by experts (lines J1-J2). ILP methods that incorporate declarative bias have the additional advantage that one can systematically and gradually restrict these general biases to incorporate domain knowledge.

Tables 9, 10, and 11 show some of the hypotheses generated by these learning systems on our dataset. Each hypothesis is labeled with the line number of the corresponding experiment from Table 8.[14] Notice that all these hypotheses are very simple; this is probably appropriate, given the small amount of data available in training.

Table 9 shows the three strictly logical hypotheses (based on ordinary Prolog, or boolean variables) that were determined to be significantly better than guessing, based on the cross-validation experiments. The simplest of these, for line F2, predicts faults whenever a derived

[12]The process for $k = 2$ took three days of compute time. However, timing comparisons are difficult since this was on a much slower machine (a 60 Mhz Sun 20) and in a different language (Prolog).

[13]The best ILP system (line C4) is statistically significantly better than either C4.5 or RIPPER with a boolean encoding, and better than C4.5 with the numeric encoding with confidence $p > 0.90$. However it is indistinguishable from RIPPER with the numeric encoding.

[14]Note that many hypotheses were generated in the course of these experiments; in the cross-validation experiments, each learning algorithm was run 10 times on different subsets of the data, yielding ten hypotheses, and we also ran each learner on the entire data, generating an eleventh variant hypothesis. We present the hypothesis resulting from running the learner on all of the data, unless otherwise noted; in general we have tried to give some feel for the actual range of hypotheses generated in the experiments.

	Learning system (options)	Errors #	Errors %	*vs.* default W-L	*vs.* default p	Time (sec) avg	Time (sec) max
	FOIL6.4	55	30.9	18-15	0.51	6,838	10,672
	(monotone)	50	28.1	20-12	0.86	11,540	16,636
	(monotone,modes)	53	29.8	8-3	0.89	35,451	63,185
$\surd$F1	FLIPPER	42	23.6	28-12	0.99	495	1,317
$\surd$F2	(monotone)	40	22.5	29-11	0.99	1,773	7,714
	(monotone,modes)	49	27.5	22-12	0.87	3,944	24,439
F3	(strong bias)	62	34.8	25-29	0.41	18	23
M1	(M=100,monotone,modes)	63	35.4	21-26	0.46	384	2,454
M2	(M=500,monotone,modes)	52	29.2	31-25	0.57	989	3,470
P1	FLIPPER/gain on proofs	58	32.6	0-0	0.00	91	100
	FLIPPER/counting clauses						
C1	(M=100,monotone,modes)	48	27.0	32-22	0.82	1,386	4,369
$\surd$C2	(M=500,monotone,modes)	41	23.0	37-20	0.97	4,737	9,917
$\surd$C3	(strong bias)	39	21.9	34-15	0.99	29	42
$\surd$C4	(biased,optimize 2x)	35	19.7	39-16	0.99	28	32
	hand-coded boolean						
I1	RIPPER	59	33.1	20-21	0.20		
I2	C4.5	52	29.2	25-19	0.64		
	hand-coded numeric						
$\surd$ J1	RIPPER	37	20.8	40-19	0.99		
J2	C4.5	44	24.7	37-23	0.93		
	generated boolean (RIPPER)						
I3	monotone, $k = 1$	47	26.4	25-14	0.93		
$\surd$ I4	monotone,$k = 2$	42	23.6	37-21	0.96		
I5	monotone,modes,$k = 1$	59	33.2	19-20	0.11		
I6	monotone,modes,$k = 2$	50	28.1	35-25	0.70		
	generated numeric (RIPPER)						
J3	monotone, $k = 1$	45	25.3	37-24	0.91		
$\surd$ J4	monotone, $k = 2$	36	20.2	41-19	0.99		
$\surd$ J5	monotone,modes,$k = 1$	37	20.8	40-19	0.99		
$\surd$ J6	monotone,modes,$k = 2$	44	24.7	32-18	0.95		

Table 8: Comparison of all learning systems on fault prediction

F1: FLIPPER with default options	faulty(C1)← mc(C1,M1,C2,M2), isa(C1,C3), ¬eq(M1,M2).
F2: FLIPPER with monotonicity constraint	faulty(C1)← mc(C1,M1,C2,M2), isa(C1,C3)
I4: RIPPER with boolean features generated from all monotone clauses of length $k \leq 2$	faulty $\Leftrightarrow (a_{1390} \wedge a_{1435} \wedge a_{1153}) \ \vee \ (a_{1355} \wedge a_{1628})$ c_{1390}: faulty(C1) ← mc(C1,M1,C2,M2),isa(C2,C3). c_{1435}: faulty(C1) ← mc(C2,M2,C1,M1),cmt(C2,M2,C1). c_{1153}: faulty(C1) ← mc(C1,M1,C1,M2),isa(C1,C3). c_{1355}: faulty(C1) ← mc(C1,M1,C2,M2),arf(C2,C1,G). c_{1628}: faulty(C1) ← mc(C2,M2,C1,M1),mrf(C3,3,C2,A2).

Table 9: Useful models: ordinary logic programs or boolean features

class calls another class; the others are variations of this. The second disjunct shown on for I4 in Table 9 is somewhat different; this disjunct has two conjunct clauses, which emphasize different types of coupling [Briand *et al.*, 1997].

Table 10 shows the hypotheses based on "counting clauses" or numerical variables that were determined to be significantly better than guessing. Most of these hypotheses predict faults based on the number of function calls (sometimes with additional constraints). Typically a class with more than 15 or so method calls is predicted to be fault prone. This is closely related to the OMMIC measure in [Briand *et al.*, 1997] which was found to be a significant fault predictor.

The hypotheses shown in Tables 9 and 10 were generated by FLIPPER or its propositional cousin, FLIPPER. Table 11 shows two hypotheses generated by FOIL and C4.5.

6 Related work

The problem finding fault-prone elements in software systems[Basili *et al.*, 1996; Chidamber and Kemerer, 1994; Briand *et al.*, 1997] has received much attention. The main contribution of this paper has been the use of ILP techniques to automatically explore a large space of possible hypotheses. This approach allows predictive models of fault density to be obtained without manually defining any numerical metrics; instead the ILP system is supplied with a high-level representation of the sample programs, expressed as a set of ground facts concerning inter-class coupling.

We have presented two novel machine learning algorithms: FLIPPER with counting clauses, and the indirect ILP approach in which numeric features were constructed based on length-k clauses. To our knowledge, no previous ILP system learns counting clauses—in fact, no previous ILP system directly measures the number of distinct proofs a clause may have, except for the special purpose of detecting determinacy. However, description logic

C2-C3: FLIPPER with monotonicity constraint, mode constraints, counting clauses, and $M = 500$, or FLIPPER with strong bias and counting clauses, optimizing once	faulty(C) $\leftarrow$ mc(C,Min1,Cout1,Mout1) $> m$ *(m ranges from 14 to 17 in different CV runs)*
C4: FLIPPER with strong bias and counting clauses, optimizing twice (typical CV run)	faulty(C) $\leftarrow$ mc(C,Min1,Cout1,Mout1) > 16 faulty(C) $\leftarrow$ mc(C,Min1,Cout1,Mout1), $\neg$isa(C,Cout1) > 32.
J1: RIPPER with hand-coded numeric features	faulty $\Leftrightarrow a_{20} \geq 15$ c_{20}: faulty(A) $\leftarrow$ mc(A,D,E,F),$\neg$frd(A,E).
J4: RIPPER with numeric features generated from all monotone clauses of length $k \leq 2$	faulty $\Leftrightarrow (a_{1325} \geq 30)$ c_{1325}: faulty(A) $\leftarrow$ mc(A,D,E,F),mc(G,F,H,I).
J5: RIPPER with numeric features generated from all monotone clauses obeying mode constraints of length $k \leq 1$	faulty $\Leftrightarrow (a_7 \geq 17)$ c_7: faulty(A) $\leftarrow$ mc(A,D,E,F).
J6: RIPPER with numeric features generated from all monotone clauses obeying mode constraints of length $k \leq 2$	faulty $\Leftrightarrow$ $(a_8 \geq 115 \wedge a_{342} \leq 0) \vee$ $(a_{365} \geq 20 \wedge a_{87} \geq 60 \wedge a_{71} \leq 497) \vee$ $(a_{94} \leq 370)$ c_8: faulty(C1) $\leftarrow$ mc(C1,M1,C2,M2). c_{71}: faulty(C1) $\leftarrow$ mc(C1,M1,C2,M2),mc(C3,M2,C4,M4). c_{87}: faulty(C1) $\leftarrow$ mc(C1,M1,C2,M2),mrf(C3,M3,C2,A2). c_{94}: faulty(C1) $\leftarrow$ mc(C1,M1,C2,M2),cmp(C1,M3,P). c_{342}: faulty(C1) $\leftarrow$ arf(C1,C2,A2),cmt(C2,M2,C3). c_{365}: faulty(C1) $\leftarrow$ arf(C1,C2,A2),cap(C1,A3,P).

Table 10: Useful models: counting clauses or numeric features

FOIL with monotonicity constraints

faulty(C1) ← mrf(C2,M2,C1,A1).
faulty(C1) ← mc(C1,M1,C2,M2), cap(C1,A1,P).
faulty(C1) ← mp(C2,M2,C1,M1).

C4.5 with hand-coded numeric features

c_{20}: faulty(C1) ← mc(C1,M1,C2,M2),¬frd(C1,C2).
c_{29}: faulty(C1) ← cmt(C1,M1,C2),¬isa(C1,C2).
c_{32}: faulty(C1) ← cmt(C1,M1,C2),¬frd(C1,C1).
c_{57}: faulty(C1) ← cmt(C2,M2,C1).
c_{76}: faulty(C1) ← arf(C2,C1,A1),¬frd(C2,C1).
c_{79}: faulty(C1) ← mrf(C1,M1,C2,A2).
c_{98}: faulty(C1) ← arf(C1,C2,A2),¬frd(C1,C2).

```
a20 <= 14 :
|   a57 <= 68 :
|   |   a51 <= 15 :
|   |   |   a29 <= 5 : NOTFAULTY
|   |   |   a29 > 5 :
|   |   |   |   a32 <= 8 : FAULTY
|   |   |   |   a32 > 8 : NOTFAULTY
|   |   a51 > 15 :
|   |   |   a79 > 11 : NOTFAULTY
|   |   |   a79 <= 11 :
|   |   |   |   a98 <= 2 : FAULTY
|   |   |   |   a98 > 2 : NOTFAULTY
|   a57 > 68 :
|   |   a57 <= 75 : FAULTY
|   |   a57 > 75 : NOTFAULTY
a20 > 14 :
|   a98 > 5 : NOTFAULTY
|   a98 <= 5 :
|   |   a51 > 17 : FAULTY
|   |   a51 <= 17 :
|   |   |   a79 <= 7 :
|   |   |   |   a76 <= 3 : FAULTY
|   |   |   |   a76 > 3 : NOTFAULTY
|   |   |   a79 > 7 :
|   |   |   |   a32 <= 16 : NOTFAULTY
|   |   |   |   a32 > 16 : FAULTY
```

Table 11: Models produced by FOIL and C4.5

learning systems have been developed that learn concepts involving number restrictions, a closely related concept [Frazier and Pitt, 1994; Cohen and Hirsh, 1994b]. A boolean version of the indirect ILP approach has been described [Cohen, 1996], but numeric features have not been previously investigated in this context.

Morasca and Ruhe [1997] have explored the use of logistic regression and rough sets to discover correlations between reliability data and a set of different possibly pertitent measures. Their approach is more aligned with our interest in hypothesis discovery, since they are not testing a specific hypothesis as embodied in a particular metric. However, ILP approaches begin with base ground atomic facts, and synthesize logic programs, whereas their techniques begin with actual numeric measures such as lines of code, rate of change etc. The precise applicability of each of these different techniques needs to be studied further; however, it seems likely that ILP methods are likely to find relevant hypotheses whenever a) the phenomena leading to faults in software elements depend on the relationships within and between the elements and b) these relationships can be derived, and encoded as ground atomic facts, by source code analysis. Our work provides an illustration of this.

7 Conclusions

In this paper, we used inductive logic programming (ILP) methods to derive hypotheses about fault density in C++ classes. These methods build hypotheses from raw relations derived from source code, and software fault data; the relations are such that various coupling metrics can be derived from them. In our experiments, a large space of possible hypotheses is searched automatically. This approach is complementary to the more typical empirical approach, in which a small number of manually derived hypotheses are tested.

Using ILP methods in this manner makes it possible to discover novel hypotheses—hypotheses that, while predictive, might not be considered by experts. In spite of the large search space considered, it is often the case that the ILP methods produce strongly predictive models.

Our experiments with the task of predicting software faults led us to uncover and correct several important deficiencies in the existing ILP learning algorithms. In particular, software fault prediction offers two significant technical challenges not present other ILP benchmark problems: there is a high level of noise, and the underlying relationships are highly non-determinate. We compared two off-the-shelf ILP systems, FLIPPER and FOIL. We discovered that FLIPPER gives significantly more accurate results than FOIL, and is also generally faster on the natural data. Experiments with synthetic data showed that FLIPPER is usually more accurate when there is noise, and faster when there is both noise and a substantial amount of indeterminacy.

Based on these experiments, we then proposed two extensions to FLIPPER. The first is an additional bias away from clauses that are empirically observed to be expensive to evaluate on the training data. This bias appears to improve run-time, at some cost in accuracy. The second extension is allows FLIPPER to learn logic programs containing *counting clauses*, a special case of the number restrictions frequently used in description logics. Such counting clauses exploit non-determinacy by using the degree of non-determinacy of a clause as an additional property upon which predictions can be based. Use of counting clauses greatly

improves FLIPPER's performance when restricted biases suggested by domain experts are used.

We also evaluated the use of ILP techniques for automatic generation of boolean and numeric metrics from the calling tree representation, the numeric features being generated using a novel variant of a process suggested by theoretical results. These features also led to accurate predictors of fault density.

Acknowledgments

The authors are grateful to Waćelio Melo for generously providing his data on fault density. We are also grateful to Philip Fong for suggesting this approach.

A The Artificial Dataset

Each dataset contains 200 training examples and 1000 test examples. The background knowledge contains 25 binary relationships, intended to model coupling relationships, named $r_1, \ldots, r_{25}$. The parameter k determines the amount of non-determinacy, and β determines the noise rate.

To generate the background knowledge, for each instance e and each relation `ri`, pick an integer k_i uniformly at random in $\{0, \ldots, k\}$. Then repeat the following k_i times: pick a second instance e' uniformly at random, and assert $r_i(e, e')$. This models random coupling relationships that have a maximal non-determinacy of k.

To assign labels to the instances, for each instance e, flip two biased coins that yield "heads" with probability $1/\sqrt{3}$. If coin 1 yields heads, then assert $r_1(e, e)$; otherwise, retract $r_1(e, e)$ (if it is true). If coin 2 yields heads, then assert $r_2(e, e')$ for some e' picked uniformly at random; otherwise, retract all facts $r_2(e, x)$ for any x. The result of this is that the condition $\exists x : \ r_1(e, e) \wedge r_2(e, x)$ is now true with probability 1/3. This target concept is similar to one of the more compact and accurate hypotheses discovered by the learner. The instance e is labeled positive if this condition is true, and negative otherwise. Finally, noise is added to the training data (but not the test data) by inverting the label of e with probability β.

B The Statistical Test

In many of the experiments discussed above, we compared two learning systems by using McNemar's test on the holdout data from a cross-validation experiment. This test is strictly speaking non-standard, as McNemar's test assumes that each trial is independent, which is not the case in this setting. While there is no dependence among the test instances, the predictions are dependent, because each prediction is a function of both the test instance and the hypothesis, and the hypotheses are obtained from overlapping (and hence dependent) sets of training examples. Following recent practise in experimental machine learning (*e.g.*, [Dietterich, 1998]), we experimentally evaluated the effect of these dependencies on quality of the statistical test. In particular, we measured the rate of *Type I errors*. A Type I error

is when the null hypothesis is incorrectly rejected; in our paper, a Type I error would lead to an incorrect claim that two learning algorithms performed differently on the fault density prediction problem.

We also used the artificial data to evaluate the accuracy of the test. Using $k = 1$, we ran FLIPPER 21 times each with $\beta = 0.2$, 0.25, and 0.3, and compared the result to the default hypothesis. The average value of the z statistic[15] of the cross-validated version of the McNemar's test is slightly *less* than the average value of the z statistic for McNemar's test on the first 200 of the 1000 independent test cases; also the cross-validated version of McNemar's test made only two Type I errors at the 95% confidence level.[16]. Both of these observations suggest that cross-validated version of McNemar's test has an acceptable rate of Type I error on data of this sort.

References

[Adé *et al.*, 1995] Hilda Adé, Luc de Raedt, and Maurice Bruynooghe. Declarative bias for general-to-specific ILP systems. *Machine Learning*, 20(1/2):119–154, 1995.

[Basili *et al.*, 1996] V. Basili, L. Briand, and W. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 1996. To appear. Also available as TR, UMD-CSD, number CS-TR-3443.

[Briand *et al.*, 1993] L. Briand, S. Morasca, V. Basili, Measuring and Assessing Maintainability at the End of High Level Design, In *Proc. Conference on Software Maintenance. Montreal, Canada: IEEE, 1993.*

[Briand *et al.*, 1997] L. Briand, P. Devanbu, and W. Melo. Defining and validating coupling measures in object-oriented systems. In *Proc. of International Conference on Software Engineering*, Boston, MA, 1997. To appear.

[Chidamber and Kemerer, 1994] S. R. Chidamber and C. F. Kemerer. A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20(6):476—493, 1994.

[Cohen and Hirsh, 1994a] William W. Cohen and Haym Hirsh. The learnability of description logics with equality constraints. *Machine Learning*, 17(2/3), 1994.

[Cohen and Hirsh, 1994b] William W. Cohen and Haym Hirsh. Learning the CLASSIC description logic: Theoretical and experimental results. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourth International Conference (KR94).* Morgan Kaufmann, 1994.

[Cohen, 1994a] William W. Cohen. Grammatically biased learning: learning logic programs using an explicit antecedent description language. *Artificial Intelligence*, 68:303–366, 1994.

[15] The z statistic is used in the normal approximation to the binomial test.

[16] At the 95% confidence level, one would expect about three of these in 63 trials.

[Cohen, 1994b] William W. Cohen. Pac-learning nondeterminate clauses. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, Seattle, WA, 1994.

[Cohen, 1995a] William W. Cohen. Fast effective rule induction. In *Machine Learning: Proceedings of the Twelfth International Conference*, Lake Tahoe, California, 1995. Morgan Kaufmann.

[Cohen, 1995b] William W. Cohen. Learning to classify English text with ILP methods. In Luc De Raedt, editor, *Advances in ILP*. IOS Press, 1995.

[Cohen, 1995c] William W. Cohen. Pac-learning recursive logic programs: efficient algorithms. *Journal of AI Research*, 2:501–539, May 1995.

[Cohen, 1996] William W. Cohen. Learning with set-valued features. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Portland, Oregon, 1996.

[Devanbu and Eaves, 1994] P. Devanbu and L. Eaves. Gen++ software (See `http://seclab.cs.ucdavis.edu/~devanbu/genp` for more information).

[Dietterich, 1998] Thomas G. Dietterich. Approximate statistical tests for comparing supervised classification learning algorithms. *Neural Computation*, 10(7), 1998.

[Džeroski *et al.*, 1992] Sašo Džeroski, Stephen Muggleton, and Stuart Russell. Pac-learnability of determinate logic programs. In *Proceedings of the 1992 Workshop on Computational Learning Theory*, Pittsburgh, Pennsylvania, 1992.

[Fenton, 1992] N. Fenton. *Software Metrics*. Chapman and Hall, 1992.

[Frazier and Pitt, 1994] M. Frazier and L. Pitt. Classic learning. In *Proceedings of the Seventh Annual ACM Conference on Computational Learning Theory*, New Brunswick, NJ, 1994. ACM Press.

[Fürnkranz and Widmer, 1994] Johannes Fürnkranz and Gerhard Widmer. Incremental reduced error pruning. In *Machine Learning: Proceedings of the Eleventh Annual Conference*, New Brunswick, New Jersey, 1994. Morgan Kaufmann.

[Kietz and Morik, 1991] Jörg-Uwe Kietz and Katharina Morik. Constructive induction of background knowledge. In *Proceedings of the Workshop on Evaluating and Changing Representation in Machine Learning (at the 12th International Joint Conference on Artificial Intelligence)*, Sydney, Australia, 1991. Morgan Kaufmann.

[Kietz and Wrobel, 1992] Jorg-Uwe Kietz and Stephan Wrobel. Controlling the complexity of learning in logic through syntactic and task-oriented models. In *Inductive Logic Programming*. Academic Press, 1992.

[Lavrač and Džeroski, 1992] Nada Lavrač and Sašo Džeroski. Background knowledge and declarative bias in inductive concept learning. In K. P. Jantke, editor, *Analogical and Inductive Inference: International Workshop AII'92*. Springer Verlag, Daghstuhl Castle, Germany, 1992. Lectures in Artificial Intelligence Series #642.

[Lavrač and Džeroski, 1994] Nada Lavrač and Sašo Džeroski. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, Chichester, England, 1994.

[MacGregor, 1991] R. M. MacGregor. The evolving technology of classification-based knowledge representation systems. In John Sowa, editor, *Principles of semantic networks: explorations in the representation of knowledge*. Morgan Kaufmann, 1991.

[Michalski *et al.*, 1986] R.S. Michalski, I. Mozetic, J. Hong, and N. Lavrac. The multipurpose incremental learning system AQ15 and its application to three medical domains. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, Philadelphia, Pennsylvania, 1986. Morgan Kaufmann.

[Morasca and Ruhe, 1997] Sandro Morasca and Günther Ruhe Knowledge Discovery from Software Engineering Measurement Data: A comparative Study for of Analysis techniques. In *Proc SEKE 97*, 1997.

[Muggleton and Feng, 1992] Stephen Muggleton and Cao Feng. Efficient induction of logic programs. In *Inductive Logic Programming*. Academic Press, 1992.

[Pagallo and Haussler, 1990] Giulia Pagallo and David Haussler. Boolean feature discovery in empirical learning. *Machine Learning*, 5(1), 1990.

[Pazzani and Kibler, 1992] Michael Pazzani and Dennis Kibler. The utility of knowledge in inductive learning. *Machine Learning*, 9(1), 1992.

[Quinlan and Cameron-Jones, 1993] J. Ross Quinlan and R. M. Cameron-Jones. FOIL: A midterm report. In Pavel B. Brazdil, editor, *Machine Learning: ECML-93*, Vienna, Austria, 1993. Springer-Verlag. Lecture notes in Computer Science # 667.

[Quinlan, 1990] J. Ross Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3), 1990.

[Quinlan, 1994] J. Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann, 1994.

[Quinlan, 1995] J. Ross Quinlan. MDL and categorical theories (continued). In *Machine Learning: Proceedings of the Twelfth International Conference*, Lake Taho, California, 1995. Morgan Kaufmann.

[Shapiro, 1982] Ehud Shapiro. *Algorithmic Program Debugging*. MIT Press, 1982.

[Srinivasan *et al.*, 1996] A. Srinivasan, S. H. Muggleton, R. D King, and M. J. E. Sternberg. Theories for mutagenicity: a study of first-order and feature based induction. *Artificial Intelligence Journal*, 85(1,2):277–299, 1996.

[Weiss and Kulkowski, 1990] Sholom Weiss and Casmir Kulkowski. *Computer Systems that Learn*. Morgan Kaufmann, 1990.

[Woods and Schmolze, 1992] W. A. Woods and J. G. Schmolze. The KL-ONE family. *Computers And Mathematics With Applications*, 23(2-5), March 1992.