A Comparative Study of Inductive Logic Programming Methods for Software Fault Prediction

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

Prem Devanbu
AT&T Labs–Research
Florham Park, NJ 07932
prem@research.att.com

Abstract

We evaluate inductive logic programming (ILP) methods for predicting fault density in C++ classes. In this problem, each training example is a C++ class definition, represented as a calling tree, and labeled as "positive" iff faults (*i.e.*, errors) were discovered in its implementation. We compare two ILP systems, FOIL and FLIPPER, and explore the reasons for their differing performance, using both natural and artificial data. We then 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.

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 significant 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, represented as a calling tree, and labeled as "positive" or "negative" according to whether faults (*i.e.*, errors) were discovered in its implementation.

ILP methods are a natural choice for program classification for several reasons. First, the examples have a natural structure which can be easily extracted, given modern program analysis tools [Devanbu and Eaves, 1994]. This naturally suggests methods which, like ILP, can learn directly from structured examples. Second, collecting data is relatively expensive—in our case, only 122 examples were collected from a semester-long study involving an entire software engineering class. While ILP systems tend to be less efficient than propositional learning systems, this is not a substantial disadvantage in this setting; the cost of applying even a very expensive learning method would still be dominated by the cost of data collection. Third, ILP systems often allow greater deal of control over the bias of the learning system than propositional systems [Adé *et al.*, 1995; Kietz and Wrobel, 1992]—perhaps because of the relatively high cost of learning. In other domains, appropriately restricting bias has led to improved generalization accuracy in problems such as this one, where examples are scarce and some prior knowledge about the learning problem is available [Cohen, 1994a; Pazzani and Kibler, 1992].

Unfortunately, while ILP methods are natural to use on 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 ILP systems: FOIL [Quinlan, 1990; Quinlan and Cameron-Jones, 1993] and FLIPPER [Cohen, 1995b]. To summarize, FLIPPER is generally faster and achieves better results. Our conclusions, based on these experiments and further experiments with artificial data, are that the performance differences are partly due to differences in pruning strategies, and partly due to a low-level FOIL performance optimization. Specifically, in computing the information gain of a clause, FOIL counts the number of positive and negative "tuples" (alternative proofs) for a clause, rather than the number of pos-

[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.

itive and negative examples covered. We propose an alternative strategy for improving performance in non-determinate domains, based on limiting the total number of proofs that any single example may have.

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 prior knowledge is used to restrict the hypothesis space.

This paper is intended to be of primary interest to researchers studying machine learning. A companion paper will investigate certain issues that are of more immediate interest to the software engineering community, including a more detailed analysis of the impact of various types of prior knowledge, and a more extensive comparison to propositional learning approaches that rely on feature engineering.

2 BACKGROUND

2.1 THE PREDICTION PROBLEM

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.

Although 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], relatively little is known about what properties of software are important in predicting faults. 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. Based on this proposal, one of us (Devanbu) chose 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 then extracted from the source code using a program analysis tool called GEN++ [Devanbu and Eaves, 1994]. In all, 20,929 background facts for these relations concerning the 122 examples were extracted.

Although these relations allow one to easily express various sorts of coupling relationships, they also support theories that do not mention any sort of coupling; for instance, the clause "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. Table 1 also gives for each predicate a set of *suggested modes*. (Here "+" indicates a variable should be used as an input, "−" indicates an output, and "?" indicates either input or output.)

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 learning process—a means that is supported by many ILP systems.

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. We will briefly summarize FLIPPER since it is less well-known; however, a more detailed description can be found elsewhere [Cohen, 1995b]. 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 1.) 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* con-

Table 1: Relations available for training

Predicate & Mode	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 is used by member function M1 of class C1, and C1≠C2.

taining 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. 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.

After FLIPPER stops adding clauses, the predicate definition is "optimized" so as to more closely approximate the effect of conventional reduced error pruning. Clauses are considered in turn in the order in which they were added. 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.

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 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 most of the experiments described below, however, this feature 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 bias definition for the corresponding search space.

3 EXPERIMENTAL RESULTS

Results with FOIL and FLIPPER. 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, testing all learners on the same folds. The predictions made on each individual test case were then recorded, and McNemar's test was used to compare the predictions

[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 $d = 64$.

```
function IREP*(Data)
begin
  Data0 := copy(Data);
  PredDef := an empty predicate
  while ∃ positive examples∈ Data 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
       PredDef := Compress(PredDef,Data0)
       return PredDef
    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 1: The FLIPPER algorithm

made by the different learners.[4]

Table 2 summarizes our 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 (which always predicts negative) using McNemar's test; and the average and maximum run time of the ten learning trials.[5] 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 merely 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. First, we suppressed use of negative literals;[6] 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;[7] further, only FLIPPER performs statistically significantly better than the default classifier (indicated with a $\surd$ 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 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 re-

[4] To compare learners L_1 and L_2 with McNemar's test, one determines if the probability $P(L_1$ is correct $\mid L_1$ and L_2 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.

[5] On a 250MHz MIPS Irix computer with 8Gb of main memory.

[6] The rationale for this is that the coupling relationships describable by the background predicates should make faults more likely, rather than less likely.

[7] 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.

Table 2: Comparison of learning systems on fault prediction

	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
√F1	FLIPPER	42	23.6	28-12	0.99	495	1,317
√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
√C2	(M=500,monotone,modes)	41	23.0	37-20	0.97	4,737	9,917
√C3	(strong bias)	39	21.9	34-15	0.99	29	42
√C4	(biased,optimize 2x)	35	19.7	39-16	0.99	28	32
	hand-coded boolean						
B1	C4.5	52	29.2	25-19	0.64		
B2	RIPPER	59	33.1	20-21	0.20		
	hand-coded numeric						
N1	C4.5	44	24.7	37-23	0.93		
√N2	RIPPER	37	20.8	40-19	0.99		

stricted bias suggested by the metrics proposed by Briand *et al* [1997].[8] This hand-coded bias also increases the error rate.

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. 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.

Analysis of 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. In particular, the predicates `mc`, `mp`, `mrf`, and `arf` can be instantiated many different ways for some examples, even when used in their suggested modes. This means that a long clause containing many of these 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 example may have. 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. The intent is that 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].

Lines M1-M2 of Table 2 show the performance of FLIPPER with modes and with M set to 100 and 500.

[8] This bias generates 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`).

Table 3: Comparison of FOIL and FLIPPER on artificial data

	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	***	***	***

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.[9] Unfortunately, the restriction also adversely affects accuracy. The increase in error for $M = 100$ versus no restriction, for instance, 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.

Analysis of error rate differences. The differences in error rate between FLIPPER and FOIL with the standard settings are also of interest. 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. Instead, FOIL counts positive and negative "tuples"—essentially, the number of distinct proofs of all positive (respectively negative) examples. 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. One would expect this effect to be most serious domains that are both highly indeterminate and noisy; the former condition being when proof counts and example counts will differ most, and the latter condition being when 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 2 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.

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. (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 3 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,[10] 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. This seems to happen because FOIL often overfits noisy data and constructs many long clauses, which in non-determinate domains are very expensive.

We also compared the error rates of standard FLIPPER with the proof-counting variant. Of nine non-zero differences, standard FLIPPER has a lower error rate eight times, a statistically significant difference.[11] This again supports the hypothesis that counting

[9] 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 unnecessary in binding the variables in L, and then evaluates a simplified clause in which these literals are deleted. Unfortunately deleting such literals changes the number of distinct proofs of a clause.

[10] With $p > 0.99$ on a two-tailed signed test.

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

proofs rather than examples is inadvisable in non-determinate noisy domains.

Evaluation of 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 non-standard, as McNemar's test assumes that each trial is independent, which is not the case in this setting. We also used the artificial data to evaluate the accuracy of this 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[12] 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[13]. Both of these observations suggest that cross-validated version of McNemar's test has an acceptable rate of Type I error.

4 ILP WITH NUMBER RESTRICTIONS

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 3 CHILD))` in the description logic CLASSIC, whereas in Prolog it would be written as the possible substitutions for X for the query

$$\leftarrow \text{person}(X),\ \text{child}(X,Y1),\ \text{child}(X,Y2),\ \text{child}(X,Y3),\ Y1\neq Y2,\ Y2\neq Y3,\ Y1\neq Y3$$

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.

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

[13] A Type I error is when the null hypothesis is incorrectly rejected. (The independent test examples were used to determine if the null hypothesis was correct.) At the 95% confidence level, one would expect about three of these in 63 trials.

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 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 $k = 3$ covers an example $p(e)$ iff e has at least 3 children: p(X) $\leftarrow$ 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;[14] 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 2. 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 $M = 100$ ($p > 0.95$).

We conclude by describing two additional experiments. 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 modest improvement in error rate, giving the lowest error rate we have achieved so far on this problem.

The second 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 B1-B2 summarize the results for the boolean encoding, and lines N1-N2 summarize the numeric encoding.

Again, the numeric encoding improves performance in both cases—significantly for RIPPER ($p > 0.99$).[15] This result can be viewed as a propositional analog to the improvement gained by extending FLIPPER with counting clauses.

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.[16] Similar results have been found elsewhere [Srinivasan *et al.*, 1996]. The benefit of general-purpose ILP methods is illustrated by lines F1-F2, in which good generalization performance (not statistically worse than the best results) is obtained with a minimum of feature engineering. ILP methods that incorporate declarative bias have the additional advantage that one can systematically and gradually restrict these general biases to incorporate domain knowledge.

5 CONCLUSIONS

We investigated the use of ILP methods for the task of predicting fault density in C++ classes. While these methods are natural to use on this task, this problem offers significant technical difficulties not present other ILP benchmark problems; in particular, 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 show that FLIPPER is usually more accurate when there is noise, and faster when there is both noise and a substantial amount of indeterminacy.

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 runtime, 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.

Acknowledgments. The authors are grateful to Waćelio Melo for generously providing his data on fault density.

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

[14]The dependence analysis optimization was disabled, for the same reasons described above.

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

[16]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.

at random, and assert $r_i(e, e')$. This models random coupling relationships that have a maximal nondeterminacy 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.

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.*, 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.

(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.

(Devanbu and Eaves, 1994) P. Devanbu and L. Eaves. Gen++ manual, version 1.2 (send mail to `genserver-@research.att.com` for more information). Technical report, AT&T Bell Laboratories, November 1994.

(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.

(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.

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

(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.

(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.

(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.