LIME: A System for Learning Relations

Eric McCreath[*] and Arun Sharma[**]

Department of Artificial Intelligence, School of Computer Science and Engineering,
The University of New South Wales, Sydney NSW 2052, Australia,
Email : {ericm,arun}@cse.unsw.edu.au

Abstract. This paper describes the design of the inductive logic programming system LIME. Instead of employing a greedy covering approach to constructing clauses, LIME employs a Bayesian heuristic to evaluate logic programs as hypotheses.
The notion of a *simple clause* is introduced. These sets of literals may be viewed as subparts of clauses that are effectively independent in terms of variables used. Instead of growing a clause one literal at a time, LIME efficiently combines simple clauses to construct a set of gainful candidate clauses. Subsets of these candidate clauses are evaluated via the Bayesian heuristic to find the final hypothesis.
Details of the algorithms and data structures of LIME are discussed. LIME's handling of recursive logic programs is also described.
Experimental results to illustrate how LIME achieves its design goals of better noise handling, learning from fixed set of examples (and from only positive data), and of learning recursive logic programs are provided. Experimental results comparing LIME with FOIL and PROGOL in the KRK domain in the presence of noise are presented. It is also shown that the already good noise handling performance of LIME further improves when learning recursive definitions in the presence of noise.

1 Introduction

This paper is a progress report on LIME—an inductive logic programming (ILP) system that induces logic programs as hypotheses from ground facts.[1] Unlike many systems (e.g., Quinlan's FOIL [23], Muggleton and Feng's GOLEM [19], and Muggleton's Progol [21]) that employ a greedy covering approach to constructing the hypothesis one clause at a time, LIME employs a Bayesian framework that evaluates logic programs as candidate hypotheses. This framework, introduced in [14] and incorporated in the LIME system, has been shown to have the following features:

– better noise handling,

[*] Supported by an Australian Postgraduate Award and by the Australian Research Council grant A49703068 to Ross Quinlan.
[**] Supported by the Australian Research Council grant A49530274.
[1] This is a preliminary report of work in progress; updated versions of the report on LIME can be obtained from `http://www.cse.unsw.edu.au/~ericm/lime` .

- ability to learn from fixed example size (i.e., there is no implicit assumption that the distribution of examples received by the learner matches the true "proportion" of the underlying concept to the instance space),
- capability of learning from only positive data[2], and
- improved ability to learn predicates with recursive definitions.

Empirical evidence was provided for the effectiveness of this framework with respect to the above four criteria in [14]. The present paper describes the design of the LIME system. Since the Bayesian heuristic employed in LIME requires evaluation of entire logic programs as hypotheses, the search space is naturally huge. This paper explains how LIME exploits the structure in the hypothesis space to tame the combinatorial explosion in the search space.

The main idea of the design of LIME is that instead of growing a clause one literal at a time, it builds candidate clauses from "groups of literals" referred to as "simple clauses". These simple clauses can be very efficiently combined to form new clauses in such a way that the coverage of a clause is the intersection of the coverage of the simple clauses from which it is formed. Once a list of candidate clauses has been constructed, the Bayesian heuristic is used to evaluate its subsets as potential hypotheses.

Structurally, LIME has four distinct stages:

- structural decomposition (preprocessing of the background knowledge),
- construction of simple clauses,
- construction of clauses, and
- search for the final hypothesis.

Within each stage care is taken that no redundant information is passed to the next stage. A diagrammatic outline of the various phases of LIME is given in Figure 1.

1.1 Related Work

We refer the reader for preliminaries and notation about ILP to the book by Nienhuys-Cheng and de Wolf [22] or the article by Muggleton and De Raedt [18]. Other source books for ILP are Bergadano and Gunetti [3], Lavrač and Džeroski [12], and Muggleton [17]. Below, we briefly touch upon work that is related to ours.

The design of LIME is somewhat reminiscent of the approach of translating an ILP problem into a propositional one. Džeroski, Muggleton, and Russell [10] describe the transformation of determinate ILP problems into propositional form. The systems LINUS and DINUS by Lavrač and Džeroski [12] employ attribute value learners after transforming restricted versions of the ILP problem.

Kietz and Lübe [11] introduced the notion of k-local clauses which is somewhat similar to simple clauses. They divided a clause into determinate and nondeterminate part and further subdivided the nondeterminate part into k-local

[2] The system can also learn from only negative data, but this capability is diminished when the concept requires a recursive definition.

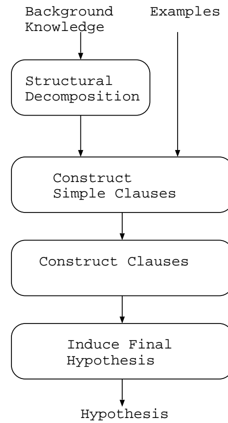

Fig. 1. The partitioning of LIME's stages.

parts. They were motivated by a desire to find efficient algorithms for subsumption.

As already noted LIME considers entire logic programs as hypotheses instead of building the hypothesis one clause at a time. Another system that follows this approach is TRACY by Bergadano and Gunetti [2]. During the preprocessing phase of the background knowledge, LIME automatically extracts type and mode information. Similar issues are addressed by Morik et al [16] in their system MOBAL.

1.2 Outline of the Paper

The outline of the paper is as follows. In Section 2, we introduce the noise model and the Bayesian framework employed in LIME. In Section 3, we describe the hypothesis language of LIME and discuss the notion of simple clauses in some detail. Sections 4–9 are devoted to a detailed discussion of the system design of LIME. Preprocessing of the background knowledge is discussed in Section 4. Section 5 describes the construction of simple clauses. Later sections describe the construction of clauses and the search for the final hypothesis. Finally, in Section 10 we report on experiments with LIME.

2 Noise model and the Bayesian Heuristic

In this section we describe our framework for modeling learning from data of fixed example size with noise. Within this framework, we derive a Bayesian heuristic for the optimal hypothesis.

Let X denote a countable class of instances. Let D_X be a distribution on the instance space X. Let $\mathcal{C} \subseteq 2^X$ be a countable concept class. Let $D_\mathcal{C}$ represent

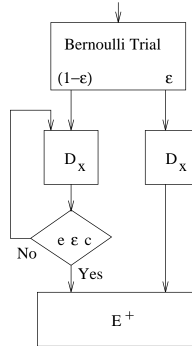

Fig. 2. Model of positive example generation

the distribution on $\mathcal{C}$. Let H be a hypothesis space and P be the distribution (prior) over H. The concept represented by a hypothesis $h \in H$ is referred to as the extension of h (written: $\text{ext}(h)$). Further, let $\mathcal{C}$ and H be such that:

- for each $C \in \mathcal{C}$, there is an $h \in H$ such that $C = \text{ext}(h)$; and
- for each $C \in \mathcal{C}$, $D_{\mathcal{C}}(C) = \sum_{\{\ h \in \mathcal{H} | C = \text{ext}(h)\ \}} P(h)$.

Let $\theta(C)$ denote the "proportion" of the concept C with respect to the instance space X, that is, $\theta(C) = \sum_{x \in C} D_X(x)$.

We assume that a concept C is chosen with the distribution $D_{\mathcal{C}}$. Let $\epsilon \in [0, 1]$ be the level of noise. Suppose we want to generate m positive examples and n negative examples (the reader should note that in the fixed example model, m and n are independent of the concept C).

Each of the m positive examples are generated as follows: With probability ϵ, a instance is randomly choose from X and made a positive example (this could possibly introduce noise). With probability $1 - \epsilon$, an instance is repeatedly selected randomly from X until the instance is an element of the concept. This instance is the positive example generated. Figure 2 illustrates this process. The generation of negative examples is done similarly[3].

A fixed example size framework allows learning to take place from only positive data (and from only negative data) in addition to the usual combination of positive and negative data. Additionally, the choice of such a framework can be motivated as follows. Many learning systems have an implicit expectation that the distribution of examples received by the learner matches the true "proportion" of the underlying concept to the instance space. However, in many situations such an assumption is unjustified. Usually the size of positive and negative examples is fixed and independent of the concept being learned. As an example, consider a learner presented with a set of 100 positive and 100 negative examples of cancer patients. It is very unlikely that this set of examples is representative of the population from where the examples are drawn.

[3] The level of noise ϵ can be made different for the positive and negative examples, but for simplicity we take it to be the same.

We now derive a Bayesian heuristic for finding the most probable hypothesis h given the example set E.[4] This induction can be formally expressed as follows.[5]

$$h_{\text{induced}} = \max_{h \in H} P(h|E) \tag{1}$$

Using Bayes' formula, $P(h|E)$ can be expressed as follows.

$$P(h|E) = \frac{P(h)P(E|h)}{P(E)} \tag{2}$$

We will apply Occam's razor in computation of $P(h)$, the prior probability of the hypothesis h, thereby assigning higher probabilities to simpler hypotheses. $P(E|h)$, probability of examples E given that hypothesis h represents the target concept, can be calculated by taking the product of the conditional probabilities of the positive and negative example sets. As each positive example is generated independently, $P(E^+|h)$ may be calculated by taking the product of the conditional probabilities of each positive example. $P^+(e|h)$, the conditional probability of a positive example e given hypothesis h, is computed as follows.

$$P^+(e|h) = \begin{cases} \frac{D_X(e)(1-\epsilon)}{\theta(\text{ext}(h))} + D_X(e)\epsilon, & \text{if } e \in \text{ext}(h); \\ D_X(e)\epsilon, & \text{if } e \notin \text{ext}(h). \end{cases} \tag{3}$$

A few words about the above equation are in order. Given that h represents the target concept, the only way in which $e \notin \text{ext}(h)$ is if the right hand path in Figure 2 was chosen. Hence, in this case the conditional probability of e given h is $D_X(e)\epsilon$. On the other hand, if $e \in \text{ext}(h)$ then either the left or right hand paths in Figure 2 could have been chosen. The contribution of the right hand path to $P^+(e|h)$ is then $D_X(e)\epsilon$. If the left hand path is taken, then the instance drawn is guaranteed to be from the target concept; hence $D_X(e)(1-\epsilon)$ is divided by $\theta(\text{ext}(h))$—the proportion of the target concept to the instance space. By a similar reasoning we compute $P^-(e|h)$, the conditional probability of a negative example e given hypothesis h.

$$P^-(e|h) = \begin{cases} \frac{D_X(e)(1-\epsilon)}{1-\theta(\text{ext}(h))} + D_X(e)\epsilon, & \text{if } e \notin \text{ext}(h); \\ D_X(e)\epsilon, & \text{if } e \in \text{ext}(h). \end{cases} \tag{4}$$

Now, $P(E|h)$ can be computed as follows.

$$P(E|h) = \prod_{e \in E^+} P^+(e|h) \prod_{e \in E^-} P^-(e|h) \tag{5}$$

We let TP denote the set of true positives $\{\ e \in E^+ \mid e \in \text{ext}(h)\ \}$; TN denote the set of true negatives $\{\ e \in E^- \mid e \notin \text{ext}(h)\ \}$; FPN denote the set of false positives and false negatives, $\{\ e \in E^+ \mid e \notin \text{ext}(h)\ \} \cup \{\ e \in E^- \mid e \in \text{ext}(h)\ \}$.

[4] All references to example sets are actually references to *example multisets*. E is the union of positive (E^+) and negative (E^-) examples.

[5] The notation $\max_{h \in H} P(h|E)$ denotes a hypothesis $h \in H$ such that $(\forall h' \in H)[P(h|E) \geq P(h'|E)]$.

Substituting 3 and 4 into 5 and using TP, TN, and FPN, we get the following.

$$P(E|h) = \left(\prod_{e \in E^+ \cup E^-} D_X(e) \right) \left(\frac{1-\epsilon}{\theta(\text{ext}(h))} + \epsilon \right)^{|\text{TP}|} \left(\frac{1-\epsilon}{1-\theta(\text{ext}(h))} + \epsilon \right)^{|\text{TN}|} \epsilon^{|\text{FPN}|} \quad (6)$$

Now substituting 6 into 2 and 2 into 1 and performing additional arithmetic manipulation, we obtain the final h_{induced} as $\max_{h \in H} Q(h)$, where $Q(h)$ is defined as follows.

$$Q(h) = \lg(P(h)) + |\text{TP}| \lg \left(\frac{1-\epsilon}{\theta(\text{ext}(h))} + \epsilon \right) + |\text{TN}| \lg \left(\frac{1-\epsilon}{1-\theta(ext(h))} + \epsilon \right) + |\text{FPN}| \lg(\epsilon) \quad (7)$$

Hence, in our inductive framework, a learning system attempts to maximize $Q(h)$ (referred to as the *quality* of the hypothesis h). LIME evaluates logic programs as candidate hypotheses by computing their Q values. The details of how LIME computes $P(h)$ and $\theta(ext(h))$ are provided in Sections 8 and 9, respectively.

Finally, we would like to note that the treatment of noise in our framework has some similarities to that of Angluin and Laird [1]. Their noise level parameter measures the percentage of data with the incorrect sign, that is, elements of the concept being mislabeled as negative data and vice versa. In their model 50% noise level means the data is truly random, whereas in our model truly random data is at noise level of 100%. Thus, in their model it is not useful to consider noise levels of greater than 50%. Our current model requires that the noise level be provided to the system. Although this may appear to be a weakness, in practice, a reasonable estimate suffices, and it can be shown that with increase in the example size, the impact of an inaccurate noise estimate diminishes. It should be noted that experiments reported in this paper always used a noise parameter of 10% in computing $Q(h)$ even if the actual noise in the data was considerably higher.

3 Hypothesis Space and Simple Clauses

The hypothesis space of LIME is chosen not only to reduce the size of the search but to also simplify the structure of the search space. The hypothesis space of LIME consists of definite ordered logic programs whose clauses are:

- *function free*: this simplifies the structure of each clause without seriously affecting expressiveness,

– *determinate*[6]: although this restricts expressiveness, clause handling is simplified as each variable may only be bound in a unique way, and
– terms in the head of the target clause are required to be distinct variables, and terms in literals of the body can only be variables (this restriction can be overcome by introducing predicates that define a constant and the equality relation in the background knowledge).

We next motivate the notion of simple clauses.

Each clause consists of a head literal and a list of body literals. In general, a literal within the body of a clause may use a variable that has been introduced and bound by a previous literal in the body. Since variables within the clause overlap, this literal can't be considered independent. However, the list of literals in the body may be broken down into lists of literals that are effectively independent in terms of variables used. We refer to such lists as *simple clauses*. This notion is best illustrated with the help of an example. Consider the predicate `teen_age_boy` where the background knowledge consists of each person's age, each person's sex, and the greater than relation. Now a clause[7] for `teen_age_boy` is:

$$\texttt{teen_age_boy}(A) \leftarrow \texttt{male}(A), \texttt{age}(A, B), B > 12, 20 > B.$$

The above clause can be "constructed" from the following three simple clauses by "combining" their bodies.

$$\begin{array}{l} \texttt{teen_age_boy}(A) \leftarrow \texttt{male}(A). \\ \texttt{teen_age_boy}(A) \leftarrow \texttt{age}(A, B), B > 12. \\ \texttt{teen_age_boy}(A) \leftarrow \texttt{age}(A, B), 20 > B. \end{array}$$

To see how the above works and to formally define a simple clause, we introduce the notion of a directed graph associated with an ordered clause. For any two literals l_1 and l_2 in a clause, we say that l_2 is *directly dependent* on l_1 just in case there exists a variable in l_2 that is bound in l_1. Hence, a directed graph may be associated with a clause by associating each literal to a node in the graph and by forming a directed edge from the node associated with literal l_1 to l_2 just in case l_2 is directly dependent on l_1. A literal l_2 is said to be *dependent* on literal l_1 just in case there is a path in the graph from l_1 to l_2.

Clearly, graphs associated with determinate clauses are acyclic. A literal in a clause is said to be a *source* literal just in case there are no literals in the clause on which it depends. A literal in a clause is said to be a *sink* literal just in case there are no literals in the clause that depend on it. Clearly, the head of a clause is always a source literal.

---

[6] Intuitively, a clause is *determinate* if each of its literals are determinate; a literal is *determinate* if each of its variables that do not occur in previous literals has only one possible binding given the bindings of its variables that appear in previous literals. See Džeroski, Muggleton, and Russell [10] for a more formal definition.

[7] LIME will of course use a slightly different representation as it does not allow constants; this example has been chosen to illustrate the notion of a simple clause.

Definition 1. A clause is said to be *simple* just in case it contains at most one sink literal.

The directed graph for the clause defining teen_age_boy is show in Figure 3. Since it has three sink nodes, it is not a simple clause. However, the directed graph for the clause describing over12, shown in Figure 4, has exactly one sink node and hence is a simple clause.

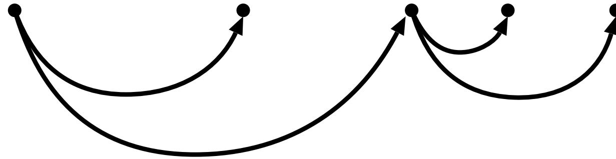

Fig. 3. Directed graph corresponding to the example clause

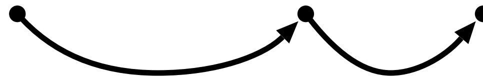

Fig. 4. Directed graph corresponding to the over twelve clauses

We next discuss some properties of simple clauses which makes them suitable for our needs. Since we are considering determinate clauses, the ordering of the literals in the body is important; the body of a clause is considered a list, not a set. Although there is some flexibility in the ordering, the constraint that "if l_1 depends on l_2 then l_1 is to the right of l_2" must be respected. This constraint yields an equivalence class over the set of clauses. If $\mathcal{C}$ is the set of all clauses, then we define $[c]$ to denote the set of all clauses in $\mathcal{C}$ that are equivalent to c. As these partitions are finite, and there is a lexicographic ordering over the clauses, a unique clause may be used to represent the equivalence partition, which we take to be the clause in $[c]$ with the least lexicographic ordering.

Let b_1 and b_2 be the bodies of two clauses with the same head. Then $b_1 \uplus b_2$ denotes the concatenation of b_1 with b_3, where b_3 is b_2 with all the b_1 literals removed. Consistency in variable naming is carefully maintained as follows:

– variables in the heads of the two clauses are the same;

– if there is a literal in b_1 and another in b_2 with the same predicate symbol and the same variables that have been bound in previous literals, then the new variables in these two literals have the same variable names. A simple way of maintaining this consistency in variable naming is to name variables by the simple clauses that created them.

We also adopt the notation $\uplus$ for clauses. Let $c_1 = h_1 \leftarrow b_1$ and $c_2 = h \leftarrow b_2$. Then $c_1 \uplus c_2$ is $h_1 \leftarrow b_1 \uplus b_2$ if $h_1 = h_2$ and undefined otherwise.

Simple clauses have three useful properties. First, any clause may be constructed from a finite set of simple clauses. Second, the intersection of the coverage of a set of simple clauses is the coverage of the clause formed by combining the set of simple clauses. The third property is about the completeness of the method of constructing simple clauses; i.e., there is an algorithm that enumerates the complete set of simple clauses for a given hypothesis language.

The next three propositions formalize these properties. Let $\mathcal{C}$ be the set of all clauses in the hypothesis space and let $\mathcal{S}$ be the set of all simple clauses. Clearly $\mathcal{S} \subset \mathcal{C}$.

Proposition 1. *For all $c \in \mathcal{C}$, there exists a finite set of simple clauses S such that $c \in [\uplus_{s \in S} s]$.*

Proof. Let $c = h \leftarrow b$ and let g be the graph associated with c. Then for each sink literal in g we construct a simple clause by including all the literals that this sink literal is dependent on. Let $\{h \leftarrow b_1, h \leftarrow b_2, \dots, h \leftarrow b_n\}$ be the set of simple clauses thus formed.

We claim that $c \in [h \leftarrow b_1 \uplus b_2 \uplus \cdots \uplus b_n]$. Clearly, $b_1 \uplus b_2 \uplus \cdots \uplus b_n$ will not contain any literal not in c as each literal in the body of the simple clauses is from c. Also, the combined clause, $h \leftarrow b_1 \uplus b_2 \uplus \cdots \uplus b_n$, will not be missing any literal from c because each literal in c is either a sink literal or has at least one sink literal dependent on it. In the former case the literal will be found in the simple clause formed by the sink literal; in the latter case the literal will be found in the corresponding simple clause. □

We next show that the coverage of a clause can be calculated by taking the intersection of the coverage of its simple clauses. This is because in the case of determinate clauses, the variable bindings for the simple clauses and the corresponding combined clause match. Hence, given the same interpretation prescribed by the background knowledge, if all the literals in the combined clause are true then all the literals in the simple clauses will also be true, and vice versa.

Proposition 2. *Let $c \in \mathcal{C}$ be given. Let the set of simple clauses $\{s_1, s_2, \dots, s_n\}$ be such that $c = s_1 \uplus s_2 \uplus \cdots \uplus s_n$. Then the coverage of c is the intersection of the coverage of each simple clause in $\{s_1, s_2, \dots, s_n\}$.*

The above property yields a very efficient method of calculating coverage of a clause — by taking the conjunction of coverage bit vectors associated with the simple clauses that are combined to form the clause.

Proof. Suppose $c \in \mathcal{C}$ covers an instance e. When e is resolved with c, each variable in c is uniquely bound (since c is determinate) in such a way that each literal in the body of c is true in the interpretation implied by the background knowledge. As each simple clause contains both a sink literal and all the literals that this sink literal depends on, the binding for each variable in the simple clause will be the same as the binding in c. Hence, each literal in the body of the simple clause will also be true in the intended interpretation. Thus, each simple clause also covers the instance e.

Suppose an instance e is covered by each simple clause in the $\{s_1, s_2, \ldots, s_n\}$. Then for each s_i there is a unique variable binding (since each s_i is also determinate) σ_i that witnesses coverage of e by s_i. Moreover, each literal in the body of $s_i\sigma_i$ is true in the intended interpretation. Now, the same variables may appear in different simple clauses. It is easy to argue that when e is resolved with different simple clauses the binding for a variable appearing in these simple clauses is the same (if this was not the case then we will have a contradiction to the assumption that c is determinate). Hence, when the bodies of the simple clauses are combined the bindings for variables across all the simple clauses will be the same. Therefore, each literal in c will also be true in the interpretation. Hence, c covers e. □

Proposition 3. *There exists an algorithm that enumerates the complete set of simple clauses.*

Proof. We first discuss the idea behind such an algorithm. The graph associated with a simple clause contains one sink literal which directly or indirectly depends on all other literals in the clause. Now if this sink literal is removed from the simple clause, we get a clause that has one or more sink literals. For each sink literal in this new clause, a new simple clause may be created by including the sink literal and all the literals that the sink literal depends on. Each new clause thus formed is simple and is smaller than the original simple clause. Reversing this process, it is easy to see that any simple clause may be created by combining a set of smaller simple clauses with a new literal l in such a way that the newly formed clause has l as the only sink literal. The one exception to this property is the simple clause with empty body. A complete algorithm for enumerating the simple clauses follows directly from this property, and such an algorithm forms the basis of LIME's simple clause table construction.

Algorithm 1 *Simple Clause Enumeration.*

begin
  current_simple_clause := $\{h \leftarrow .\}$
  output $h \leftarrow .$
  loop do
    $N := \{\}$
    foreach $S \subset$ *current_simple_clause* **do**
      foreach $l \in$ *possible_literals* **do**
        sc := combine l with S

```
      if sc is a new simple clause then
        output sc
        N := N ∪ {sc}
      fi
    od
  od
  current_simple_clause := current_simple_clause ∪ N
od
end
```

We now show that Algorithm 1 enumerates the complete set of simple clauses. The proof is by induction: we show that after the i'th iteration of the loop all simple clauses with up to i literals in their bodies have been enumerated.

Clearly this is the case for $i = 0$ as the only simple clause with 0 literals in its body is the clause with empty body.

Now suppose the inductive hypothesis is true for $i = k$. Then all simple clauses with k or fewer literals in their bodies will be in '*curent_simple_clause*'. After the next iteration of the loop all simple clauses with $k+1$ literals in their bodies will have been enumerated. This is because any simple clause with $k+1$ literals can be formed by a set of simple clauses with at most k literals in their bodies, and a new literal. Hence, the inductive proposition is true for $i = k+1$. Also note that the number of new simple clauses in each iteration of the loop is finite, as the number of both possible subsets and new literals are finite. □

The above algorithm is clearly very inefficient in the way it forms simple clauses — in each iteration it repeatedly considers simple clauses that have already been formed, and it also must detect and remove clauses that are not simple. By requiring new simple clauses to contain a variable from the previous level, repetition in the algorithm is removed; and by maintaining the literal dependency information, the non-simple clauses may be avoided. These techniques are employed by LIME.

4 Preprocessing the Background Knowledge

The first stage in LIME's inductive process involves the preprocessing of the background knowledge. This phase has three goals:

- automatic extraction of information from the background knowledge to enable the system to dynamically direct the search,
- removal of any redundancy within the background knowledge, and
- encoding of the background knowledge so that it may be efficiently indexed for the search.

We briefly discuss these aspects of the preprocessing phase.

4.1 Extracting Type and Mode Information

Each term in a predicate has an implicit type. A clause in which the type associated with a variable is inconsistent will not form part of a good hypothesis. Hence, by learning the type information from the examples and the background knowledge, inconsistent clauses may be skipped in the search. LIME uses a flat type hierarchy. Integer and floating point types are inferred simply from the syntax. Other types are induced from the example and the background knowledge. This process is thoroughly examined in [13].

Also, since the search space is restricted to determinate clauses, it is useful to know the mode restrictions for each predicate prior to the search. In the absence of such information each time a literal is added to a clause the system would need to assert that unbound variables are uniquely bound. As this check is essentially the same each time it is conducted, considerable improvement in performance can be achieved if mode information was available. LIME extracts mode information from the data which enables it to skip clauses that are not determinate. This process is also detailed in [13]. Another system that addresses these issues is MOBAL [16].

4.2 Removing Redundancy

There are three ways in which redundancy is removed from the background knowledge. First, if a set of relations are equivalent then only one needs to be considered in the inductive process. For this purpose two relations are said to be equivalent if they consist of identical ground facts in such a way that the predicate name and the ordering of the terms in the predicate are ignored. Second, if there exists symmetry within the terms in a relation then it is only necessary to consider one ordering of the terms. Consider the `add` relation which is symmetric in the first two terms. If the variables in the first two terms of an `add` literal in the body of a clause are flipped the new clause will be equivalent with respect to its coverage and size. Hence, only one of the clauses need be considered. This is illustrated in the two `mult` clauses shown below. Although, they are different syntactically, they may be considered equivalent, and hence only one needs to be considered in the search space.

$$\texttt{mult}(A,B,C) \leftarrow \texttt{inc}(D,A), \texttt{mult}(D,B,E), \texttt{add}(E,B,C).$$
$$\texttt{mult}(A,B,C) \leftarrow \texttt{inc}(D,A), \texttt{mult}(D,B,E), \texttt{add}(B,E,C).$$

Third, as only determinate logic programs are considered, any background relations that do not produce a determinate clause are not considered in the search space.

4.3 Improving Indexing

Once the background knowledge is preprocessed it is used either to determine a ground query or a query that uniquely binds new variables. Due to the large

number of queries to be performed, this operation must be efficient. Hence, hash tables are use to store the background knowledge. The time complexity of a query operation is $O(1)$ with respect to the number of ground facts defining the background predicate.

5 Simple Clause Table Construction

After the background knowledge is preprocessed, a table of candidate simple clauses is constructed. The simple clause table is the central data structure in LIME. Care is taken that there are no repetitions in this table. Also, as the table is constructed a record is maintained of both the instances each clause covers and the binding of each variable for different instances. This makes the process more efficient as the variable bindings need not be recalculated each time a simple clause is extended to form a new simple clause. The simple clause table of LIME may be viewed as consisting of two tables:

- *Simple Clause Coverage Table*: The first part of the table contains information about the coverage of instances by candidate simple clauses. This part of the table is stored as a bit vector: 1 indicating that the simple clause covers the instance and 0 indicating that the simple clause does not cover the instance.[8] The advantage of this storage scheme is that the coverage of a clause formed by combination of two simple clauses can be very efficiently determined by taking the conjunction of the bit-vectors describing the coverage of the two simple clauses.
- *Variable Binding Table*: The second part of the table consists of the binding information for each variable introduced in the simple clauses. Since we are concerned here with only determinate clauses, these bindings are unique. We use X to represent the fact that a variable does not have a binding for the instance.

An example of LIME's simple clause table is shown in Figure 5. This table captures the snapshot when LIME is in the process of learning the `add` relation. The two tables, *Simple Clause Coverage Table* and the *Variable Binding Table*, are shown side by side. It can be seen that the base clause for the `add` relation may be formed by combining the simple clauses 30 and 33. The conjunction of the bit-vectors for the these clauses will give the coverage of the following definite clause:

$$\texttt{add}(V0, V1, V2) : -\texttt{zero}(V0), \texttt{equal}(V0, V2).$$

The above clause once formed can be disjuncted with the clause consisting of just one simple clause (no. 500) to form the complete definition of the `add`

[8] In addition to 0 and 1, we also use the *don't care* state, denoted here as X, to indicate that a better simple clause exists (hence, the coverage information is not tabulated, although the simple clause possibly produces a useful new variable, and hence is kept).

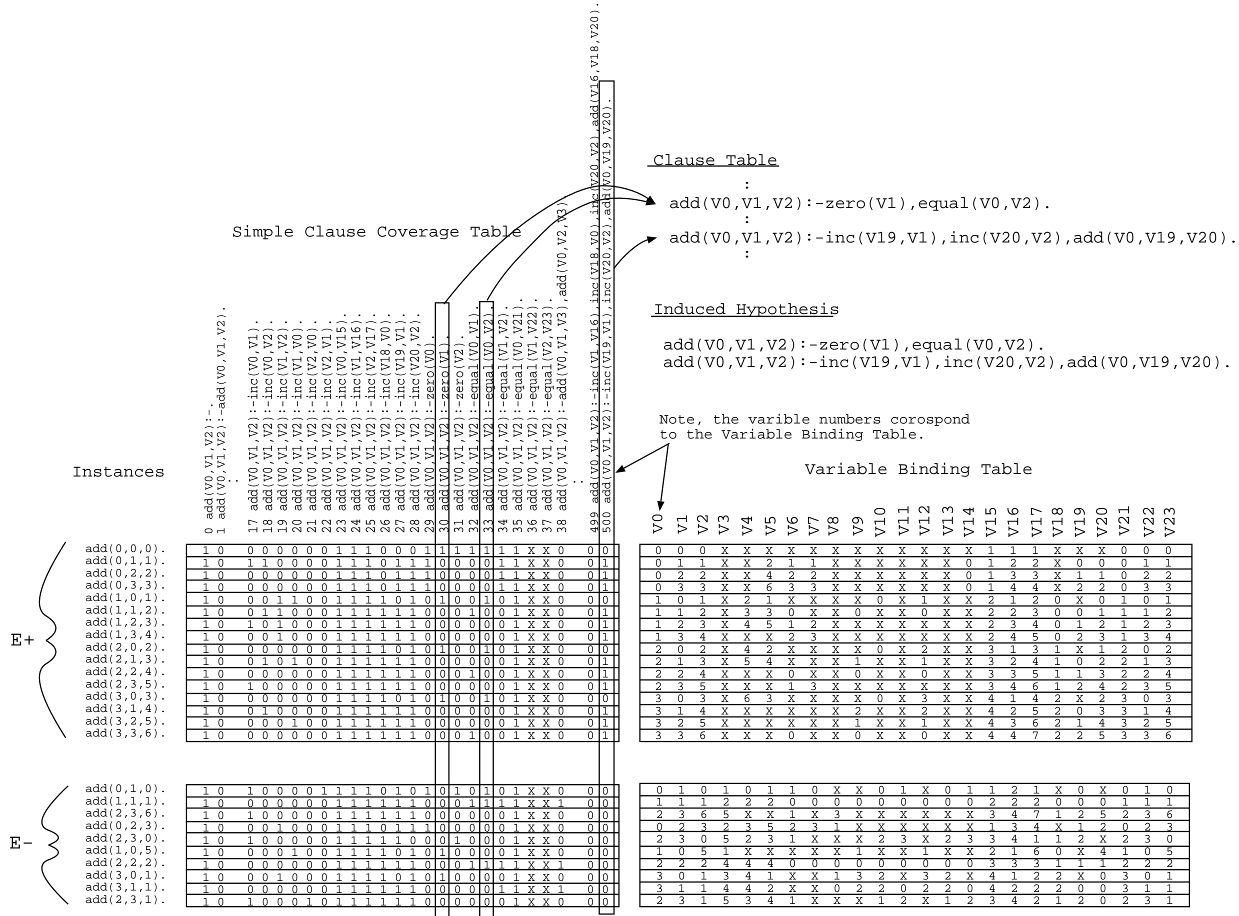

Fig. 5. Tables constructed in the simple clause stage for *add*

relation. The reader should note that disjunction of the bit vectors for the simple clause 500 with the conjunction of the bit vectors for 30 and 33 covers all the positive instances and none of the negative instances.

The simple clause table acts as an intermediate stage between the enhanced background knowledge and the candidate clauses. In many respects this intermediate stage is redundant, as the system could generate the candidate clauses directly from the preprocessed background knowledge. However, there are at least four efficiency reasons for doing so: removal of redundancy, introduction of memorization, removal of paths from search space that are not fruitful, and consolidating the construction of new variables in a clause at an initial stage.

First, we discuss the reduction of redundancy. Suppose a clause $c_1 = h \leftarrow l_1, l_2, l_3, l_4.$ consists of three simple clauses:

$$\begin{array}{l} h \leftarrow l_1, l_2. \\ h \leftarrow l_3. \\ h \leftarrow l_4. \end{array}$$

By constructing the simple clauses first and then forming c_1 by combining them, c_1 is only considered only once. However, if the system constructed c_1 literal by literal, the same clause c_1 many be considered many times as outlined below.

$$\begin{array}{llll} h \leftarrow l_1. \Rightarrow & h \leftarrow l_1, l_2. \Rightarrow & h \leftarrow l_1, l_2, l_3. \Rightarrow & h \leftarrow l_1, l_2, l_3, l_4. \\ h \leftarrow l_2. \Rightarrow & h \leftarrow l_2, l_3. \Rightarrow & h \leftarrow l_2, l_3, l_1. \Rightarrow & h \leftarrow l_2, l_3, l_1, l_4. \\ h \leftarrow l_4. \Rightarrow & h \leftarrow l_4, l_3. \Rightarrow & h \leftarrow l_4, l_3, l_2. \Rightarrow & h \leftarrow l_4, l_3, l_2, l_1. \\ \vdots & \vdots & \vdots & \vdots \end{array}$$

However, the above redundancy could be eliminated without the intermediate stage. One way to do this would be to place a syntactic ordering on the literals, and adhering to this ordering in considering clauses. However, this introduces its own problem: the syntactic ordering may not be the most gainful path in constructing a clause thereby making the use of a gain heuristic in the search less effective.

Second, the simple clause table introduces memorization in the system. LIME records the coverage of simple clauses, hence each time the literals of a simple clause are considered within a clause, the coverage does not need to be recalculated, it is simply looked up in the simple clause table. This is also the case for the variable bindings.

Third, the simple clause table provides a mechanism for removing entire branches of the search space. LIME only records simple clauses that are not dead-ends, thus eliminating the search of clauses that are not potentially useful in the final hypothesis. This is best illustrated with an example. Consider two literals l_1 and l_2. When these literals are combined in a clause, the clause covers no positive or negative examples[9] Any clause that contained these literal would

[9] The clause should also cover no examples used in the theta estimation (see Section 9). Otherwise, this constraint would fail when the system is learning from only negative

be a dead-end. Hence, by identifying l_1 and l_2, and not considering them, a whole series of 'dead-ends' are removed.

Fourth, by separating the clause generation into two stages, induction of simple clauses followed by induction of candidate clauses, all aspects of generating new variables in a clause are assigned to the first stage. This makes the latter stage more efficient as it is not concerned with generating new variables and maintaining their bindings.

5.1 Algorithm for Simple Clause Coverage Table and Variable Binding Table

We now present the algorithm for constructing the two tables. The algorithm maintains three data structures: a list of simple clauses, a list of bit vectors representing coverage of the simple clauses, and a table of variable bindings for the variables used in the simple clauses. These structures are initialized to contain just the simple clause with an empty body. Then candidate literals are created by considering each predicate symbol in the preprocessed background knowledge with all possible variable bindings generated until now and some new variable provided certain conditions are satisfied. A syntactic ordering is used to label variables to avoid considering literals which are equivalent to the ones already considered. If the coverage of a clause is different from the coverage of clauses generated until now or if the clause introduces a new variable, then it is incorporated into the data structure. To ensure that no simple clauses are repeated, the new literal must contain at least one variable from the previous cycle through the background relations. Note, some simple clauses produce new variables that are of use, but their instance coverage is of no value, in which case the new variables are recorded, but not the bit vector. The variable binding table not only maintains the variable binding of each variable for each instance, it also maintains an index of the simple clause the variable was generated in, and also maintains the level at which the variable was created. The task is complete if no new simple clauses are added in a given iteration, or if one of the tables is full. The detailed algorithm follows:

Algorithm 2 *Simple Clause Table Construction.*

Input :
  BG – preprocessed background knowledge
Output :
  C – A list of simple clauses
  BV – A bit vector table that stores coverage information for simple clauses
  VT – A variable binding table for each variable used in a simple clause
begin
  $C := h \leftarrow \{\}$ /* h is the head of the target predicatedefinition */

---

examples, as the target hypothesis would in general cover no negative examples and clearly no positive examples. Hence, the system would consider the target hypothesis a dead-end.

```
  Add to BV the bit vector for the empty body clause
  Add to VT the variable bindings for the empty body clause
  added := true
  level := 0
  while added do
    added := false
    foreach predicate symbol P in BG do
      foreach [ possible variable assignment σ for P
              [ with at least one variable from the previous level ] do
        lit := literal formed by P and σ
        clause := simple clause formed by adding lit to simple clauses
                    from C that originate variables from σ
        vector := generate bit vector from clause
        binding_vectors := compute the binding vector for
                    each new variable in literal
        if vector not in BV then
          /* This simple clause is not equivalent to a previous one */
          Add clause to C
          Add vector to BV
          Add (binding_vectors, level) to VT
          added := true
        else if binding_vectors may be useful then
          Add clause to C
          Add (binding_vectors, level) to VT
          added := true
        fi
        if any table full then
          break while loop
        fi
      od
    od
    level := level + 1
  od
  output C, BV, VT
end
```

6 Clause Table Construction

The simple clauses may now be combined to form a table of candidate clauses. Clearly, this has to be done efficiently as there are 2^k candidate clauses to consider for any k simple clauses. To this end LIME takes advantage of the following.

- Simple clauses may be combined efficiently by conjunction of bit vectors.
- When combining simple clauses, large branches of the search tree may be pruned without explicit search.

– The selection heuristic feeds back clear bounds on the required search.

We next shed some light on the crucial aspects of the clause table construction algorithm, followed by the details of the actual algorithm.

The algorithm is essentially depth-first search in nature.[10] At each node in the search tree the clause associated with the node is considered in conjunction with every simple clause. A gain heuristic is employed to direct which portion of the search space to consider. The resulting clauses are added to the best candidate clause list. There is a restriction on the number of candidate clauses maintained at any given time; only the best candidates added to the list are saved. Also, at each node in the search tree a list of the most gainful clauses are generated. Size of this list is also restricted and is dependent on the level in the search tree — deeper one goes in the search tree, smaller is the list of gainful clauses. This is because the heuristic is less accurate high up the search tree. Overlap is eliminated in the search by placing an ordering on the simple clauses, and requiring that they be considered in that order. This ensures that each simple clause is only considered once.

Figure 6 shows a search tree with $gain_list_length = 100$, which is the default value. It should be noted how the branching decreases as the tree depth increases. Suppose the depth of the tree is 5, then if there was no restriction of the branching factor, there will be 10101010101 nodes[11]. However, by reducing the branching factor as the tree is descended, the number of nodes is reduced to 10101 nodes, thereby making the search feasible without significantly affecting the chance of finding optimal clauses.

There are two stopping criteria for the search to bound the depth of each branch in the search tree. As simple clauses are combined the new clauses cover fewer examples. Thus, an obvious stopping criterion is when the clause covers no example. The other stopping criterion is when a node is reached such that the best possible clause that can be derived from descendents of this node are not good enough to make it to the candidate clause list.

Finally, a few words on the gain heuristic employed to guide the search for simple clauses and the Bayesian heuristic employed to determine the most gainful clauses. Suppose *old_clause* and *new_clause* are two clauses with the obvious meaning. Then the gain heuristic for going from *old_clause* to *new_clause* is calculated as $(n_old - n_new) \times (\lg(p_new + 2))$, where n_old denotes the size of the negative coverage of *old_clause*, n_new denotes the size of the negative coverage of *new_clause*, and p_new is the size of the positive coverage of *new_clause*. The calculation of the Bayesian heuristic for the most gainful clause is based on the idea of Q-heuristic from Section 2.

The algorithm is given below. It is a standard implementation of the depth-first algorithm using the program stack recursively calling **probe_simple_clause**. Note that **calc_gain_list_size** calculates the number of branches in the search tree at a certain depth in the tree. Also, **gain** estimates the gain when a simple

[10] Earlier implementations have looked at best-first [5, 24] and depth-bounded discrepancy search [25], but depth-first was found to be the most effective.

[11] This is a decimal number

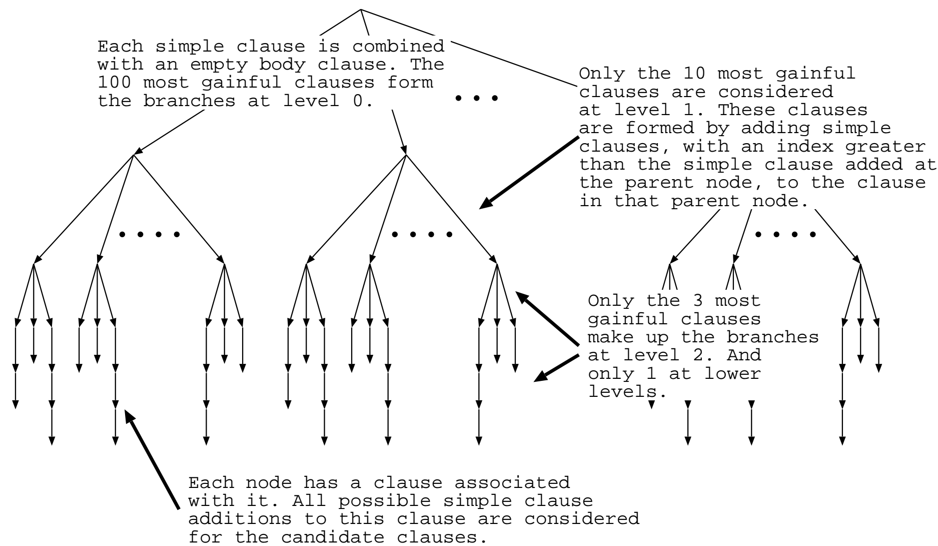

Fig. 6. The search tree where gain_list_length = 100

clause is combined with a clause to form a new clause. And **best_bay** estimates the best possible value for the Q-heuristic given this clause forms part of the final hypothesis. This best estimate is achieved by assuming the other clauses that make up the final hypothesis, which is a set of clauses, is as good as this clause in terms of positive cover, accuracy, and prior probability.

A list of clauses' data structure and associated functions has been created for this algorithm. This list is sorted by a value associated with each clause. Also, the cover of each clause is maintained in the form of a bit vector. For the gain list, the index of the last simple clause added to the clause is kept with each clause in the list. When a new list is created it is given a maximum size. If a list has reached its maximum size and a clause is added which has a value worse than the worst clause in the list, then the clause is ignored and the list is not altered. If a clause is added which has a value better than the worst clause in the list, then the worst clause is removed and the added clause is inserted in its appropriate place.

Algorithm 3 *Clause Construction from Simple Clauses.*

Input :

S – Array of simple clauses and their coverage bit vectors

/* Each element in S consists of the clause body */

/* and a bit vector giving the coverage of the simple clause */

Output :

C – A set of candidate clauses.

Parameters :

```
  num_candidate_clauses /* The maximum number of candidate */
                        /* clauses to generate */
  gain_list_length /* The length of the gain list used at level 0 */
begin
  foreach simple clause c ∈ S  do
    compute gain(simple clause with empty body, c)
  od
  Sort S in descending order of gain computed above
  Let C be a list of size num_candidate_clauses
  Let E be a record representing a new clause
  Initialize E as follows
  E.body := {}
  E.bitvector := all bits set to 1
  E.last_index := −1
        /* last_index is the index of the last */
        /* simple clause in S used to form the new clause */
  probe_simple_clause(S, C, E, 0)
  output  C
end


probe_simple_clause(S, C, R, L)
  Let GL be a new list of size calc_gain_list_size(L, gain_list_length)
  foreach i ∈ {R.last_index + 1, R.last_index + 2, ... , |S| − 1}  do
    Let NC be a record representing a new clause
    Initialize NC as follows
    NC.body := S[i].body ⊎ R.body
    NC.bitvector := S[i].bitvector ∧ R.bitvector
    NC.last_index := i
    if (NC covers some elements) ∧ (NC.bitvector ≠ R.bitvector)  then
      gain := gain(R, NC)
      if [ ( extensions to NC could prove better than           ]  then
         [ the worst clause in C) ∨ (C list not full)           ]
        Add (NC, gain) to the list GL
      fi
      value := bay_best(NC)
      Add (NC, value) to the list C
      /* Of course, the above addition only takes place if there is */
      /* space in C or the Bayesian estimate for NC */
      /* is better than the clause with the worst estimate in C */
    fi
  od
  foreach GE ∈ GL  do
    probe_simple_clause(S, C, GE, L + 1)
  od

calc_gain_list_size(level, length)
```

return $\lfloor length^{(2^{-level})} \rfloor$

gain(*old_clause*, *new_clause*)

$n_old :=$ negative_cover(*old_clause*)
$n_new :=$ negative_cover(*new_clause*)
$p_new :=$ positive_cover(*new_clause*)
return $(n_old - n_new) \times (\lg(p_new + 2))$

best_bay(clause)

$NN :=$ number of negatives
$NP :=$ number of positives
$TP := NP$
$TN := NN -$ negative_cover(*clause*)
$FP := 0$
$FN :=$ negative_cover(*clause*)
$S := \frac{NP}{\text{positive_cover}(clause)}$
theta := estimate_theta_cover(*clause*)
return $TP \times (\lg(\frac{1-noise}{theta} + noise)) +$
$(NN - FN \times S) \times (\lg(\frac{1-noise}{1-theta} + noise)) +$
$(FN \times S) \times (\lg(noise)) +$
$\text{prob}(clause) \times S$

6.1 Other approaches

Earlier versions of LIME employed two other search strategies. However, they turned out to be less effective than depth-first search. The first of these was the best-first search. While this approach combines the advantages of both depth-first and breadth-first by extending the search tree at the node that appears to be the most promising, it also retains the main drawback of breadth-first search — excessive storage requirement. This may partly be overcome by maintaining a bounded set of nodes for exploration in the search tree. However, such an implementation tends to either not search deep enough, or to only search a small portion of the possible branches at the top of the search tree depending on the heuristic used for determining the most promising nodes. In either case the algorithm does not perform very well in many situations. For similar reasons a beam search like the one employed by Clark and Niblett [8] in CN2 is not as effective as depth search.

The other approach attempted was a depth-bounded discrepancy search [25]. This simply re-orders a depth-first search, examining more probable clauses first. Hence, the search space is restricted earlier in the search as "good" clauses are found earlier. However, because this technique required either revisiting nodes or storing the search tree, the efficiency gain did not overcome the overhead of either revisiting nodes or storage.

6.2 Efficiency

It is difficult to determine the exact actual size of the search tree as it is dynamically pruned. However, there is an upper bound on the size of the search tree. The number of nodes in the search tree depends on the *gain_list_size*.

Let $l = \lceil -\lg \frac{1}{\lg gain_list_size} \rceil$ and let $b = gain_list_size$. Then the maximum number of nodes in a tree of depth d will be

$$1 + b + b \times b^{2^{-1}} + b \times b^{2^{-1}} \times b^{2^{-2}} + \cdots +$$
$$b \times b^{2^{-1}} \times \cdots \times b^{2^{-(l-1)}} + (d-l) \times b \times b^{2^{-1}} \times \cdots \times b^{2^{-l}}$$

The number of operations required at each node is dependent linearly on the number of examples for the bit vector operation times the number of simple clauses considered at the node. It should be noted that the bit vector operations can be performed efficiently using the system level bitwise operations provided by most architectures. The memory requirement is minimal as it is a depth search algorithm.

7 Inducing the final hypothesis

The final stage in LIME's inductive process is similar to the previous stage, though with some crucial differences. First, the search is for a logic program from a set of candidate clauses. Second, instead of using conjunction of the coverage vectors of simple clauses, we use disjunction of the coverage vectors of clauses, as an instance may be covered by any one of the clauses. Third, as there may not always be independence between clauses (due to recursion), a Prolog interpreter is used to accurately evaluate the hypothesis cover. Fourth, as the Prolog interpreter used has no backtracking, order is important in the list of clauses induced. The details are given in the following algorithm.

Algorithm 4 *Induction of Final Hypotheses from Candidate Clauses.*
Input :
  C – array of clauses. /* Each element in the array consists of the clause */
                                        /* and a bit vector giving the coverage of the clause. */
Output :
  H – A set of induced hypotheses.
Parameters :
  num_final_hypothesis /* The maximum number of hypotheses induced */
  gain_list_length /* The length of the gain list used at level 0 */
begin
  Let H be a list of size *num_final_hypothesis*
  Let L be a record representing a new logic program
  Initialize L as follows
  L.clauses := {}
  L.bitvector := all bits set to 0
  probe_clause$(C, H, L, 0)$
  output C
end

```
probe_clause(C, H, R, L)
  Let GL be a new list of size calc_gain_list_size(L, gain_list_length)
        /* calc_gain_list_size is defined in Algorithm 3 */
  foreach i ∈ {0...|C| − 1}  do
    Let NL be a record representing a new logic program
    Initialize NL as follows
    NL.clauses := {C[i].clause} ∪ R.clauses
    NL.bitvector := C[i].bitvector ∨ R.bitvector
    if (NL covers some elements) ∧ (NL.bitvector ≠ R.bitvector)  then
      gain := gain_lp(R, NL)
      if [ ( extensions to NL could prove better
           than the worst clause in H) ∨ (H list not full) ]  then
        Add (NL, gain) to the list GL
      fi
      value := bay(NL)
      if value better than worst value in H  then
        Update NL.bitvector using Prolog interpreter and NL.clauses
            /* If NL.clauses is recursive then NL.bitvector (and value) */
            /* are only estimates; therefore, the Prolog interpreter is used */
            /* to compute the actual coverage and value. */
        value := bay(NL)
        Add (NC, value) to the list H
      fi
    fi
  od
  foreach GE ∈ GL  do
    probe_clause(C, H, GE, L + 1)
  od

gain_lp(old_clause, new_clause)
  n_old := negative_cover(old_clause)
  p_old := positive_cover(old_clause)
  n_new := negative_cover(new_clause)
  p_new := positive_cover(new_clause)
  return (lg(p_new + 2)) × (2 × (n_old − n_new) + (p_new − p_old))

bay(logic_program)
  NN := number of negatives
  NP := number of positives
  TP := positive_cover(logic_program)
  TN := NN − negative_cover(clause)
  FP := NP − positive_cover(logic_program)
  FN := negative_cover(logic_program)
  theta := estimate_theta_cover(logic_program)
  return TP × (lg((1−noise)/theta + noise))+
         TN × (lg((1−noise)/(1−theta) + noise))+
         (FP + FN) × (lg(noise))+
         prob(logic_program)
```

7.1 Recursive Logic Programs

In most cases each clause in a hypothesis may be considered independently. This allows the clauses to be be induced individually and then combined to form the final hypothesis. This is the approach taken in ILP systems employing a greedy covering strategy (e.g., [19, 7]). Unfortunately, the independence of each clause breaks down when recursion is involved because a recursive clause by itself will not cover any examples.

The common approach in learning recursive clauses is to include all the positive examples into the background knowledge. This allows the algorithm to determine coverage by using these facts to unify with the recursive literals in the body of the clause. However, this introduces many problems, different systems handle them in a variety of ways. For example, FOIL[6] induces a partial ordering on the constants, and then requires at least one term in the recursive literal to descend or ascend this ordering. This ensures that there are no loops in the recursive call. CHILLIN[27], on the other hand, requires that at least one term in the recursive literal be a proper sub-term of the corresponding term in the head of this clause. This ensures that the clause induced does not lead to infinite recursion.

Since, in the induction of the final hypothesis LIME considers entire logic programs, a Prolog interpreter is used to accurately determine the coverage of potential hypotheses. This step, without including the positive examples into the background knowledge, weeds out any poorly constructed recursive hypotheses. For example, if a recursive logic program is missing its base case, it will quickly be shown to cover no examples, and hence give a poor posterior probability. However, it should be noted that when LIME needs to evaluate coverage of simple clauses or clauses in previous stages, it behaves like other ILP systems and estimates the coverage of an individual recursive clause by including the positive examples in the background knowledge. However, to address the problem of infinite recursion, it does not directly restrict undesirable clauses, rather it constructs a graph of how the positive examples recursively use each other. This approach enables a better estimate of a clause's final coverage as part of a complete logic program.

This process is best explained with an example. Consider the recursive clause $add(A, B, C) \leftarrow add(B, A, C)$. We wish to estimate its coverage when it forms part of a complete hypothesis, of which we do not know the other clauses. Also suppose the positive examples consist of three instances: $\mathrm{add}(1, 2, 3)$, $\mathrm{add}(1, 1, 2)$, and $\mathrm{add}(2, 1, 3)$. Now if we naively include the positive examples into the background knowledge, then resolve the clause with each instance, the clause would cover all three instances. This clearly misrepresents the quality of the clause. So as each instance is resolved a graph is maintained of the instances an instance uses for its resolution. If a loop in the graph is detected by resolving an instance, then the recursive clause has an infinite recursive loop. Hence, one of the examples should be a base case to solve this dilemma so that the current instance is considered not covered by the recursive clause and the dependence is kept loop free. This will give a better estimate of the coverage of the recursive clauses.

Figure 7 illustrates this process for our example. At stage 1 the first instance add(1, 2, 3) is tested by resolving it with the recursive clause which uses the third instance add(2, 1, 3). Since no loops are created in the graph the first example is considered covered by the clause and the edge is added to the graph. At stage 2 the second instance add(1, 1, 2) when resolved requires itself and would form a circuit in the graph, so the clause is considered not to cover the instance, and the graph is left unchanged. At stage 3 the third instance add(2, 1, 3) would, when resolved, require the first instance add(1, 2, 3). This would form a circuit in the graph so the recursive clause could not cover both instances. One instance needs to be covered by another clause, hence, we estimate the recursive clause to still cover the first instance but not the third. Finally, the recursive clause is estimated to cover only the first of three instances. A naive approach would have it covering all three instance, which is a poor estimator of the recursive clause. Figure 7 shows the stages LIME undertakens in this process.

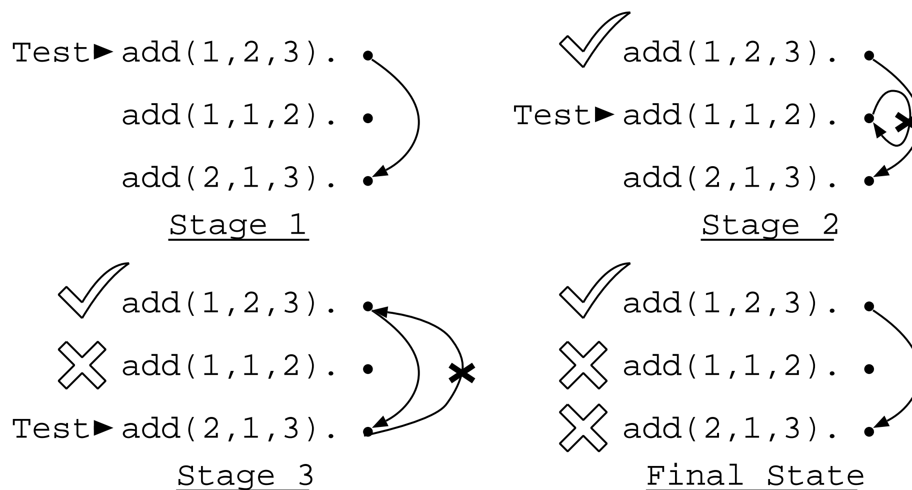

Fig. 7. Example of estimating coverage of a individual recursive clause

Also this approach directs the search toward the base cases by attempting to cover examples the recursive clause will finally cover in the logic program.

Another problem that is encountered when learning recursive logic programs is a sparse set of positive examples. That is, if many of the examples are separated by more than one resolution step then the strategy of including the positive examples in the background knowledge will not help in determining coverage. LIME partly handles this problem by the use of a Prolog interpreter in the final stage of induction. Although, a recursive clause must show some potential when evaluated by itself to form part of the candidate clauses; this will not be the case when the example set is sparse.

8 Prior Probability

To generate priors over the hypothesis space a probabilistic context-free grammar is used. A probabilistic context-free grammar $\langle G, \mathcal{P} \rangle$ is a context-free grammar G where each production rule is assigned a probability p_i. Note, $0 \leq p_i \leq 1$ and the sum of the probabilities with the same nonterminal left-hand side is 1. The probability of a derivation is given by the product of the probabilities of the productions used in the derivation. The probability of a sentence, generated by the grammar, is the sum over all possible distinct derivations from the start non-terminal to the sentence.

In many respects the way priors are attached is arbitrary. As the number of examples increases the prior becomes irrelevant. Basically, the probabilistic context-free grammar forms a mapping between sentences of the grammar, which are logic programs, and a probability value for the sentence, which is then defined to be the prior for this logic program.

Another way of attaching priors would be to encode the hypothesis into a bit string, then calculate the prior from its length. In this approach care must be taken when encoding the logic program, as it requires a prefix code. This still forms a mapping between the logic program and its probability via a bit string. By using probabilistic context-free grammars the intermediate stage is removed, simplifying the task and allowing more flexibility in the formation of the mapping.

The probabilistic context-free grammar [26] used by LIME for calculating priors of logic programs is given in table 1. The non-terminals LP, C, B, L,

P_1	$p_1 = \frac{1}{\mu_C+1}$	$LP \longrightarrow \epsilon$
P_2	$p_2 = 1 - \frac{1}{\mu_C+1}$	$LP \longrightarrow CLP$
P_3	$p_3 = 1$	$C \longrightarrow \text{head} \leftarrow B$
P_4	$p_4 = \frac{1}{\mu_L+1}$	$B \longrightarrow \epsilon$
P_5	$p_5 = 1 - \frac{1}{\mu_L+1}$	$B \longrightarrow L, B$
P_6	$p_6 = \frac{1}{n_L}$	$L \longrightarrow \text{name}_1(\overbrace{T, T, \cdots, T}^{arity_1})$
P_7	$p_7 = \frac{1}{n_L}$	$L \longrightarrow \text{name}_2(\overbrace{T, T, \cdots, T}^{arity_2})$
$\vdots$	$\vdots$	$\vdots$
P_{5+n_L}	$p_{5+n_L} = \frac{1}{n_L}$	$L \longrightarrow \text{name}_{n_L}(\overbrace{T, T, \cdots, T}^{arity_{n_L}})$
P_{5+n_L+1}	$p_{5+n_L+1} = \frac{1}{\mu_V+1}$	$T \longrightarrow v$
P_{5+n_L+2}	$p_{5+n_L+2} = 1 - \frac{1}{\mu_V+1}$	$T \longrightarrow T'$

Table 1. Grammar use by LIME for calculating prior of hypothesis.

and T correspond respectively to a logic program, clause, the body of a clause, literal, and term. μ_C is a parameter which sets the expected number of clauses in a logic program. Similarly μ_L is the expected number of literals in the body of clauses. And μ_V is the expected variable number for any term in a logic program. Details such as "commas" and "periods" are ignored as they only have cosmetic effects.

From this grammar the stochastic expectation matrix M many be calculated, given in table 2, where an element at the row corresponding to non-terminal X and the column corresponding to non-terminal Y is the expected number of times X will be replaced by Y in exactly one production rule. As the spectral radius $\rho(M)$, which is the modulus of the largest eigenvalue, is always less then 1 the probabilistic grammar is consistent [4]. That is, the sum over all the sentences generated from this grammar is 1.

$$M = \begin{array}{c} \\ LP \\ C \\ B \\ L \\ T \end{array} \begin{array}{c} \begin{array}{ccccc} \text{LP} & \text{C} & \text{B} & \text{L} & \text{T} \end{array} \\ \begin{bmatrix} 1-\frac{1}{\mu_C+1} & 1-\frac{1}{\mu_C+1} & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1-\frac{1}{\mu_L+1} & 1-\frac{1}{\mu_L+1} & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{n_L}\sum_{i=1}^{n_L} arity_i \\ 0 & 0 & 0 & 0 & 1-\frac{1}{\mu_V+1} \end{bmatrix} \end{array}$$

Table 2. Stochastic expectation matrix M for the logic program probabilistic context-free grammar.

The calculation of $P(h)$ in LIME is trivial, as there is a unique derivation from the start non-terminal LP to any logic program h. This derivation is simply obtained by parsing h and calculating the product of the probabilities assigned to the production rules. As LIME requires $\lg(P(h))$ rather than $P(h)$, $\lg(P(h))$ is calculated directly. This moves the lg into the calculation, changing multiplication to addition. Also, the numbers used are manageable — they do not become exponentially small.

Note, if any clause is added to a hypothesis then the prior must decrease, hence, for all h and c $P(h) > P(h \cup \{c\})$. This gives a simple way of calculating a bound on the maximum prior for any partly completed hypothesis. Although obvious, this bound is useful in restricting the search space.

9 θ Estimation

An estimate of $\theta(ext(h))$ is required for computing the Q heuristic for a hypothesis h. In general, LIME considers many candidate hypotheses, so an efficient method of estimating the θ value for a hypothesis is essential to make LIME

viable. Recall from Section 2 that the θ value is a measure of the proportion of the instance space a hypothesis covers.

It may be calculated exactly by taking the sum over the probability of each instance that is in the extension of the hypothesis as shown in equation 8. However, this is impossible for three reasons: the extension of the hypothesis is usually infinite; the instance space distribution is unknown; and as the instance space distribution is a mapping to the reals, the result given by a θ evaluation is not, in general, representable by a Turing machine, let alone computable by one. Hence, an approximate estimation of this value must be found.

$$\theta = \sum_{e \in \mathrm{ext}(h)} D_X(e) \tag{8}$$

LIME estimates θ by taking a random sample of n instances, then calculating the number c of these instances the hypothesis covers. Next, a Laplacian estimate, $\frac{c+1}{n+2}$, is used. Note, the random sample of instances is generated at the start of LIME's execution, and the same sample is used for all θ estimations in simple clause construction, clause construction, and the induction of the final hypotheses. This is for reasons of efficiency, and it keeps the estimation consistent across different hypotheses. A more general hypothesis, therefore, will always have the same or a higher θ estimation.

To generate a single random instance, each term in the instance is randomly selected by a uniform distribution over the constants that constitute the term's type. This implicitly assumes a uniform distribution over a finite restriction of the instance space, as prescribed by the ground terms given in the examples and background knowledge given to LIME. Although, this uniform distribution is in general different from the unknown distribution that generated the examples, it will still produce useful estimates of θ, as the θ estimate is used mainly to compare different hypotheses. The comparison by the θ estimate does not change greatly under transformations in the instance space distribution. That is, if $\mathrm{ext}(h_1) \subset \mathrm{ext}(h_2)$ then $\theta(h_1) \leq \theta(h_2)$ for any instance space distribution. This process is repeated until the required number of samples is generated. By default LIME uses 500 instances to make up the random sample.

10 Empirical Results

In this section we present experimental results to illustrate how LIME achieves its design goals of better noise handling, learning from fixed set of examples, and of learning recursive logic programs.

10.1 Recursive Logic Programs

Bratko's Logic Programs Examples A set of logic programs based on Ivan Bratko's book *PROLOG Programming for Artificial Intelligence* [5] have been

generated to assess ILP systems. These were obtained from UCI Machine Learning Repository [15] and given to LIME without changing the examples or background knowledge (some cosmetic changes are required to the file format to make it understandable to LIME). The examples consist of all the positive and negative examples restricted to lists of maximum length 3; also, the constant symbols are restricted to the numbers 1, 2, and 3. Table 3 shows the logic programs in question. The table also shows the background knowledge that is provided to the learner. As there is no noise in the examples the noise parameter is set to 0 for LIME. Table 4 gives the results of both LIME and FOIL on these data sets. Note, that LIME successfully induces the target hypothesis for 11 out of the 16 logic programs whereas FOIL was successful on 8 out of the 16 logic programs.

Quick Sort The relation quick sort is used as a bench mark to test the ability of an ILP system. Quick sort is a difficult recursive relation to learn as the key recursive clause is complex. The complexity is due to the presence of two recursive literals in the body, and the size of this clause. Another, difficulty, especially with regard to LIME, is that the recursive clause is one big simple clause which has a depth of 3, hence, LIME must explore the space of simple clauses deep enough to discover this clause. However, LIME successfully induced the following logic program for quick sort:

$$\begin{aligned}\texttt{qsort}(A,B) \leftarrow\ & \texttt{partition1}(A,C), \texttt{partition2}(A,D),\\ & \texttt{concat}(C,D,B), \texttt{partition2}(B,D).\\ \texttt{qsort}(A,B) \leftarrow\ & \texttt{partition1}(A,C), \texttt{partition2}(A,D),\\ & \texttt{qsort}(D,E), \texttt{qsort}(D,F), \texttt{concat}(E,F,B).\end{aligned}$$

LIME took 393.57 seconds for inducing the above program.

10.2 Noise

We present results from three sets of representative experiments that compare LIME with FOIL and PROGOL. The first experiment considers learning the recursive predicate `add` with different levels of noise. The second experiment is performed on the complex *krk* domain, also with different noise levels. Third, we randomly generate a domain and consider how the number of clauses in the target concept affects predictive accuracy.

Plus Two We first demonstrate LIME's superior noise handling ability for the simple concept `plus2`, which may be represented by the following logic program:

$$\texttt{plus2}(A,B) \leftarrow \texttt{inc}(A,C), \texttt{inc}(C,B).$$

In the above `inc` denotes the increment predicate available as background knowledge. A random selection of 50 positive and 50 negative examples are given to LIME. These examples include noise. The predictive error of the induced hypothesis is measured against a noise-free test set that is generated by taking the

Name	Logic Program	Background Knowledge
Concatenation	$\texttt{conc}(A,B,C) \leftarrow \texttt{empty}(A), \texttt{equal}(B,C).$ $\texttt{conc}(A,B,C) \leftarrow \texttt{components}(A,D,E),$ $\texttt{components}(C,D,F),$ $\texttt{conc}(E,B,F).$	`components`, `member`, `empty`, `equal`
Delete	$\texttt{del}(A,B,C) \leftarrow \texttt{components}(B,A,C).$ $\texttt{del}(A,B,C) \leftarrow \texttt{components}(B,D,E), \texttt{del}(A,E,F),$ $\texttt{components}(C,D,F).$	`components`, `member`, `conc`, `last`
Dividelist	$\texttt{dividelist}(A,B,C) \leftarrow \texttt{empty}(A), \texttt{empty}(B), \texttt{empty}(C).$ $\texttt{dividelist}(A,B,C) \leftarrow \texttt{odd}(A), \texttt{components}(A,D,E),$ $\texttt{dividelist}(E,F,C)$ $\texttt{components}(B,D,F).$ $\texttt{dividelist}(A,B,C) \leftarrow \texttt{even}(A), \texttt{components}(A,D,E),$ $\texttt{dividelist}(E,B,F),$ $\texttt{components}(C,D,F).$	`components`, `member`, `conc`, `last`, `del`, `insert`, `sublist`, `permutation`, `even`, `odd`, `reverse`, `palindrome`, `shift`, `subset`
Evenlength	$\texttt{even}(A) \leftarrow \texttt{empty}(A).$ $\texttt{even}(A) \leftarrow \texttt{components}(A,B,C),$ $\texttt{components}(C,D,E), \texttt{even}(E).$	`components`, `member`, `conc`, `last`, `del`, `insert`, `sublist`, `permutation`
Insert	$\texttt{insert}(A,B,C) \leftarrow \texttt{del}(A,C,B).$	`components`, `member`, `conc`, `last`, `del`
Last	$\texttt{last}(A,B) \leftarrow \texttt{components}(B,A,C), \texttt{empty}(C).$ $\texttt{last}(A.B) \leftarrow \texttt{components}(B,C,D), \texttt{last}(A,D).$	`components`, `member`
Member	$\texttt{member}(A,B) \leftarrow \texttt{components}(B,A,C).$ $\texttt{member}(A.B) \leftarrow \texttt{components}(B,C,D), \texttt{member}(A,D).$	`components`, `conc`
Oddlength	$\texttt{odd}(A) \leftarrow \texttt{components}(A,B,C), \texttt{empty}(C).$ $\texttt{odd}(A) \leftarrow \texttt{components}(A,B,C), \texttt{components}(C,D,E),$ $\texttt{odd}(E).$	`components`, `member`, `conc`, `last`, `del`, `insert`, `sublist`, `permutation`
Palindrome1	$\texttt{palindrome}(A) \leftarrow \texttt{reverse}(A,B), \texttt{equal}(A,B).$	`components`, `member`, `conc`, `last`, `del`, `insert`, `sublist`, `permutation`, `even`, `odd`
Palindrome2	$\texttt{palindrome}(A) \leftarrow \texttt{empty}(A).$ $\texttt{palindrome}(A) \leftarrow \texttt{components}(A,B,C), \texttt{empty}(C).$ $\texttt{palindrome}(A) \leftarrow \texttt{components}(A,B,C), \texttt{last}(B,A),$ $\texttt{front}(C,D), \texttt{palindrome}(C,D).$	`components`, `member`, `conc`, `last`, `del`, `insert`, `sublist`, `permutation`, `even`, `odd`, `reverse`
Permutation	$\texttt{permutation}(A,B) \leftarrow \texttt{empty}(A), \texttt{empty}(B).$ $\texttt{permutation}(A,B) \leftarrow \texttt{components}(A,C,D),$ $\texttt{permutation}(D,E),$ $\texttt{insert}(C,E,B).$	`components`, `member`, `conc`, `last`, `del`, `insert`, `sublist`, `permutation`
Reverse	$\texttt{reverse}(A,B) \leftarrow \texttt{empty}(A), \texttt{empty}(B).$ $\texttt{reverse}(A,B) \leftarrow \texttt{components}(A,C,D),$ $\texttt{empty}(D), \texttt{equal}(B,A).$ $\texttt{reverse}(A,B) \leftarrow \texttt{components}(A,C,D),$ $\texttt{components}(B,E,F),$ $\texttt{last}(C,B), \texttt{last}(E,A), \texttt{front}(D,G),$ $\texttt{front}(F,H), \texttt{reverse}(G,H).$	`components`, `member`, `conc`, `last`, `del`, `insert`, `sublist`, `permutation`, `even`, `odd`, `reverse`
Shift	$\texttt{shift}(A,B) \leftarrow \texttt{components}(A,C,D),$ $\texttt{front}(B,D), \texttt{last}(C,B).$	`components`, `member`, `conc`, `last`, `del`, `insert`, `sublist`, `permutation`, `even`, `odd`, `reverse`, `palindrome`
Sublist	$\texttt{sublist}(A,B) \leftarrow \texttt{conc}(A,C,B).$ $\texttt{sublist}(A,B) \leftarrow \texttt{components}(B,C,D), \texttt{sublist}(A,D).$	`components`, `member`, `conc`, `last`, `del`, `insertshift`
Subset	$\texttt{subset}(A,B) \leftarrow \texttt{empty}(A).$ $\texttt{subset}(A,B) \leftarrow \texttt{components}(A,C,D),$ $\texttt{member}(C,B), \texttt{subset}(D,B).$	`components`, `member`, `conc`, `last`, `del`, `insert`, `sublist`, `permutation`, `even`, `odd`, `reverse`, `palindrome`, `shift`
Translate	$\texttt{translate}(A,B) \leftarrow \texttt{empty}(A), \texttt{empty}(B), \texttt{empty}(C).$ $\texttt{translate}(A,B) \leftarrow \texttt{components}(A,C,D), \texttt{means}(C,E),$ $\texttt{components}(B,E,F),$ $\texttt{translate}(D,F).$	`components`, `member`, `conc`, `last`, `del`, `insert`, `sublist`, `permutation`, `even`, `odd`, `reverse`, `palindrome`, `shift`, `means`

Table 3. Bratko's recursive logic programs

Name	LIME	Foil
Concatenation	Unsuccessful	Successful
Delete	Successful	Successful
Dividelist	Unsuccessful	Unsuccessful
Evenlength	Successful	Unsuccessful
Insert	Successful	Successful
Last	Successful	Successful
Member	Successful	Successful
Oddlength	Successful	Unsuccessful
Palindrome1	Successful	Successful
Palindrome2	Successful	Unsuccessful
Permutation	Unsuccessful	Unsuccessful
Reverse	Unsuccessful	Unsuccessful
Shift	Successful	Unsuccessful
Sublist	Successful	Successful
Subset	Successful	Unsuccessful
Translate	Unsuccessful	Successful

Table 4. Bratko's recursive logic programs - empirical results of LIME and Foil.

"first" 20 positive examples and a random selection of 20 negative examples. This process is repeated 100 times to calculate the average predictive error. This is repeated with different noise levels and the results are shown in Figure 8. The error bars in the figure indicate the sample standard deviation. The results show that LIME is able to correctly learn the concept with noise levels of up to approximately 70%. The same test is carried out with FOIL and PROGOL.[12]

LIME performs better than FOIL and PROGOL for noise levels of up to approximately 70%. Here, FOIL over-generalizes inducing a less predictive hypothesis. This is mainly due to the covering approach which introduces unnecessary clauses. However, for noise levels greater than 70%, all three systems perform poorly.

Addition LIME's noise handling ability is demonstrated in the context of `add` (the addition relation)—a target predicate that requires a recursive definition. The target concept may be represented by the hypothesis:

$$\begin{aligned}&\texttt{add}(A,B,C) \leftarrow \texttt{equal}(A,C), \texttt{zero}(B).\\ &\texttt{add}(A,B,C) \leftarrow \texttt{inc}(D,B), \texttt{add}(A,D,E), \texttt{inc}(E,C).\end{aligned}$$

We take a random selection of 200 positive and 200 negative examples but perform only 20 repetitions at each noise level. Figure 9 shows the relationship between noise and predictive error measured against a noise-free test set of the "first" 25 positive examples and a random set of 25 negative examples. The results show that the gap between LIME and other systems widens further when the target concept requires a recursive definition. Experiments with FOIL and PROGOL were limited to 40% and 15% noise levels respectively because the quality of the programs output by these systems beyond these noise levels were difficult to assess.

[12] All our experiments are with FOIL, version 6.3 and with PROGOL, version 4.1.

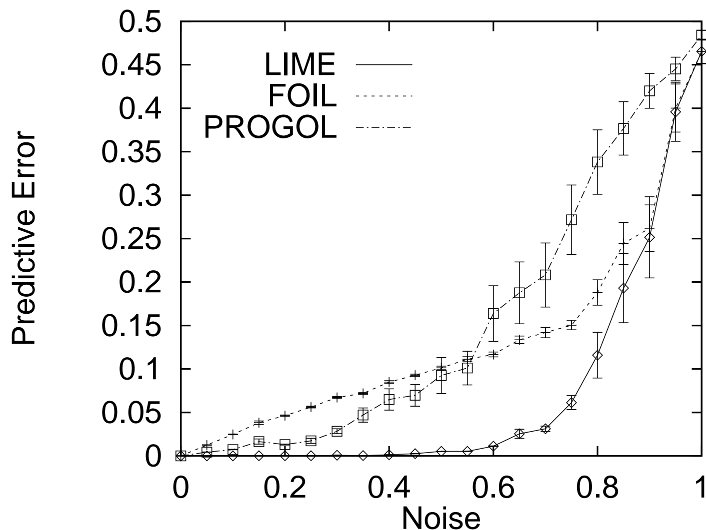

Fig. 8. Predictive Error vs Noise for `plus2`

KRK Domain The KRK domain has been well studied especially with respect to noise [20, 9]. This concept is exactly representable in first order logic, though the representation requires several clauses. The relation *illegal* checks if an end game position is an illegal chess position.

In each trial the training example set is constructed from 300 examples of which a proportion are noisy. Examples that are not noisy are chosen by randomly selecting the rank and file of each piece, where the distribution is uniform over both rank and file, then determining if the state is illegal and labeling it appropriately. A noisy example is constructed by again randomly positioning each piece, then randomly labeling it as either illegal or not-illegal. Note, a noisy example may be correctly classified. The accuracy of a hypothesis produced by the learning system is estimated by creating 10000 random examples and calculating the proportion of these the hypothesis correctly classifies. Each trial is repeated 20 times and the mean and sample standard deviation of the accuracy is calculated.

Quinlan's decision tree learner *c4.5* is also considered in this domain. The attribute giving the rank and file distance between pieces is included to help *c4.5* represent the concept. The results of these experiments are shown in Figure 10. The error bars show the sample standard deviation.

The diagram shows that PROGOL induces a more accurate hypothesis for low levels of noise, however LIME performs better at higher levels of noise. The predictive error shown by FOIL appears to be linearly dependent on the noise in the the training set. The poorer result shown by *c4.5* is partly due to its inadequacy in representing the concept.

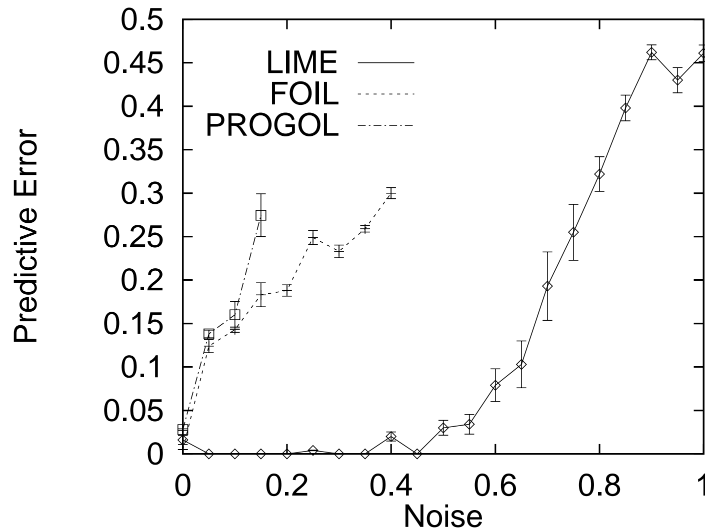

Fig. 9. Predictive error vs Noise for `add`

Randomly Generated Domain A randomly generated domain is created to examine how the ILP systems perform as the concept becomes more complex. A simple approach to introducing complexity into a concept is to include more clauses. So our measure of complexity here is the number of clauses in the target hypothesis.

Each target predicate consists of two terms, which when grounded are integers from 0 to 29. This yields an instance space of size 900. The background knowledge consists of 10 unary randomly generated predicates. Each clause consists of exactly 2 literals which are randomly selected from the background knowledge. Training and test sets are constructed by first randomly generating a target hypothesis, with the set number of clause, then this hypothesis is used to classify the 900 training instances. These are divided into training and test sets with a 90%/10% split, respectively. Before the training set is given to the learner it is corrupted by 10% noise. This process is repeated 10 times and the average error and sample standard deviation is calculated. Figure 11 shows these results for 1 to 10 clauses.

Interestingly FOIL shows the same predictive error independent of the number of clauses considered, whereas both PROGOL and LIME become less accurate as the number of clauses and hence the complexity, increases.

10.3 Learning from positive examples and from negative examples

Plus two This set of experiments gives empirical evidence that positive examples are more useful than negative examples for a target concept that is "small" with respect to the instance space distribution. The experiments also give evidence for the converse that negative examples are more useful than positive

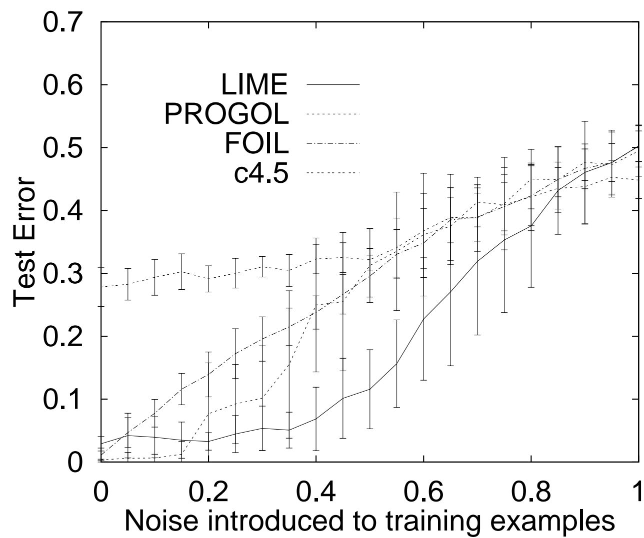

Fig. 10. The predictive error as noise is introduced into the training examples in the KRK domain.

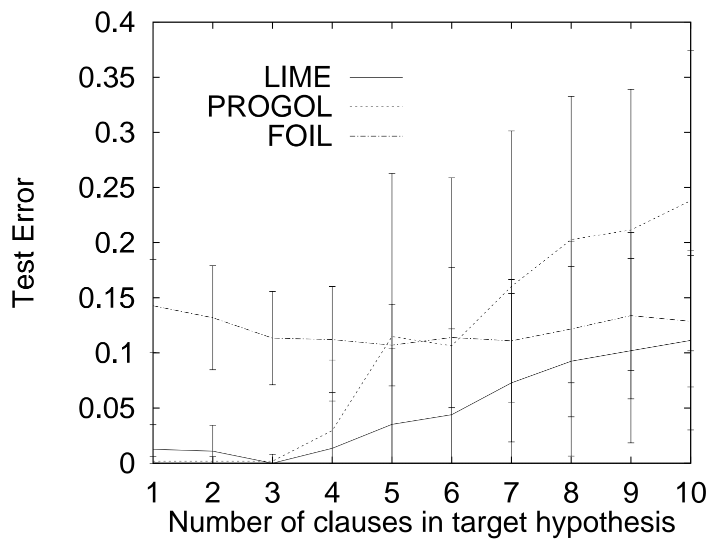

Fig. 11. The predictive error vs number of clauses in the randomly generated domain.

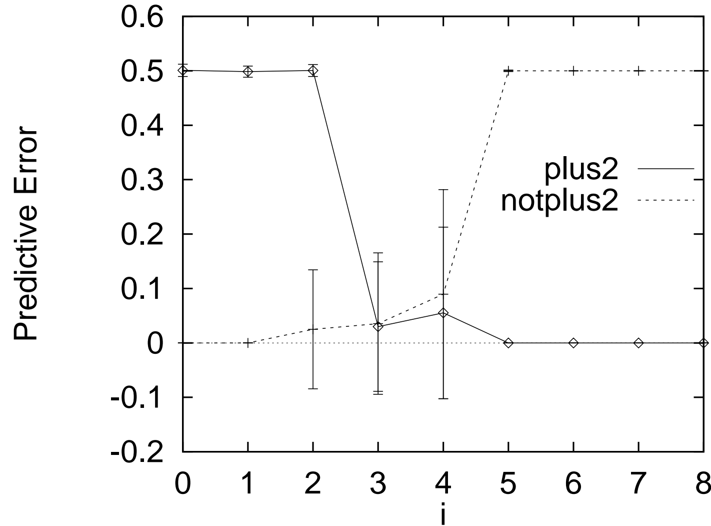

Fig. 12. Error vs i (i = number of positive examples & 8-i = number of negative examples) for the `plus2` and `notplus2` logic programs

examples for a target concept that is "large" with respect to the instance space distribution. These experiments also establish that LIME is capable of learning from only positive data and from only negative data.

We consider two concepts, the `plus2` and `notplus2` (the complement of `plus2`—that is, `notplus2`(A, B) holds if $B \neq A + 2$). It is easy to see that under reasonable assumptions, `plus2` is a "small" concept and `notplus2` is a "large" concept. Assuming the instance space $X = \{ 1..n \}^2$ then under a uniform distribution the concept covers $\frac{n-2}{n^2}$ of the instance space. In the experiment $n = 50$ and hence the `plus2` concept covers 0.0192 of the instance space. The background knowledge is identical for both `plus2` and `notplus2` consisting of the increment relation and its complement, and a constant relations for each number in the range. LIME is run on examples of `plus2` and `notplus2`. The total number of examples is invariant over each test, however, the number of positive examples is increased as the number of negative examples is decreased. Each test is repeated 100 times and the results for both `plus2` and `notplus2` are shown in Figure 12. In all experiments the test set consists of 100 randomly selected positive examples and 100 randomly selected negative examples. The error bars show the sample standard deviation.

These experiments are also repeated at different levels of noise used in generating the training examples. The graph for `plus2` and `notplus2` are shown in Figure 13.

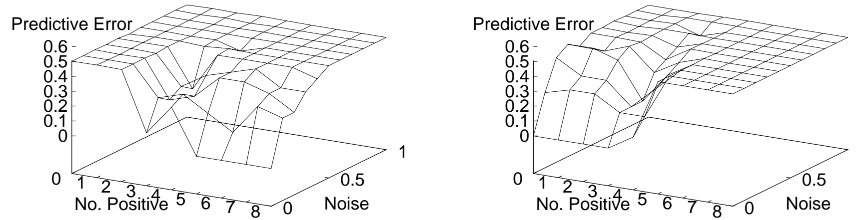

Fig. 13. Error vs i (i = number of positive examples & 8-i = number of negative examples) for the `plus2` relation shown on the left and the `notplus2` relation shown on the right. These trails are conduced with different levels of noise used in generating the training examples.

Addition In this set of experiments we examine the number of examples required for LIME to induce the *addition* relation from only positive examples. Note, as there is only positive examples the empty bodied clause is complete and consistent with respect to these examples. However, with enough positive examples the Q heuristic favors the *addition* relation over the empty body clause, as, the θ estimate for *addition* is smaller, this outweights the effect of the larger prior probability for the empty body clause.

The positive examples are randomly generated using a distribution over the instance space. Rather than restricting the instance space to a finite domain and using a uniform distribution over this space, a distribution over all three term predicates is used. The advantage of this approach is small numbers are given higher probabilities and hence occur more frequently in the example set. This aids the induction of the base case in the recursive *addition* relation. The background knowledge consists of the *inc*, *zero*, and *equal* relations these are necessary and sufficient to learn *addition*. Initially 10 positive examples are randomly generated and given to LIME, once LIME has run, the induced hypothesis is tested on the "first" 25 positive examples and a random selection of 25 negative examples. We also note if the hypothesis exactly expresses the *addition* relation. This is repeated 10 time and the mean and sample standard deviation is tabulated, also, the number of times the correct relation is induced is counted. This is repeated for different number of positive examples in increments of 10. The graph in figure 14 shows average error decreasing as the number of positive examples increases and the number of tests LIME takes to induce the exact *addition* relation.

Learning the addition relation from only negative examples is not investigated as positive examples are used in estimating the preliminary cover of recursive clauses and since these are absent in negative only data LIME will not learn recursive logic programs from only negative data.

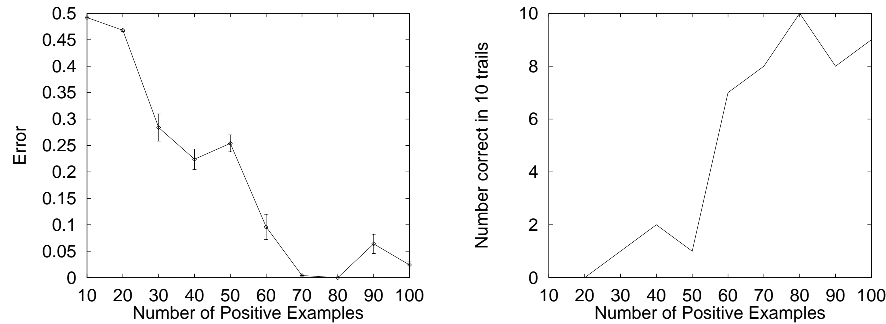

Fig. 14. LIME inducing *addition* from only positive examples. With error vs number of training examples on the left and number correctly induced hyptheses, out of 10 trials, vs number of training examples on the right.

11 Conclusion

The design of the ILP system LIME was described. The notion of simple clause was introduced and its use in the design of LIME was discussed. It was shown that combining simple clauses to form candidate clauses provides an effective alternative to growing clauses one literal at a time. Detailed algorithms for simple clause construction, clause construction, and logic program construction were given. Empirical results were presented that reinforce the superior noise handling ability of LIME. The performance of LIME is particularly good when it is learning recursive definitions in the presence of noise.

Work in progress involves application of LIME on real world domains and experiments with a boosted version of LIME.

Acknowledgements

Preliminary versions of parts of this work have had the benefit of being reviewed for earlier conferences [13, 14]. We are grateful to several anonymous reviewers for their comments.

Eric McCreath has been supported by an Australian Postgraduate Award and by the Australian Research Council grant A49703068 to Ross Quinlan. Arun Sharma has been supported by the Australian Research Council grant A49530274.

References

1. D. Angluin and P. Laird. Learning from noisy examples. *Machine Learning*, 2:343–370, 1987.
2. F. Bergadano and D. Gunetti. An interactive system to learn functional logic programs. In *Machine Learning*, 1994.

3. F. Bergadano and G. Gunetti. *Inductive Logic Programming: from Machine Learning to Software Engineering.* MIT Press, 1996.
4. T. L. Booth and R. A. Thompson. Applying probability measures to abstract languages. *IEEE Trans. Comput.*, C-22:442–450, 1973.
5. I. Bratko. *Prolog Programming for Artificial Intelligence.* Addison-Wesley, Workingham, 1990. second edition.
6. M. Cameron-Jones and J. Quinlan. Avoiding pitfalls when learning recursive theories. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence.* Morgan Kauffmann, San Mateo, CA, 1993.
7. R.M. Cameron-Jones and J.R. Quinlan. Efficient top-down induction of logic programs. *SIGART Bulletin*, 5(1):33–42, 1994.
8. P. Clark and T. Niblett. The CN2 induction algorithm. *Machine Learning*, 3(4):261–283, 1989.
9. S. Dzeroski and I. Bratko. Handling noise in inductive logic programming. In *Proceedings of the Second International Workshop on Inductive Logic Programming*, 1992. Tokyo, Japan. ICOT TM-1182.
10. S. Dzeroski, S. Muggleton, and S. Russell. PAC-Learnability of determinate logic programs. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory, Pittsburgh, Pennsylvania*, pages 128–135. ACM Press, July 1992.
11. J-U. Kietz and M. Lube. An efficient subsumption algorithm for inductive logic programming. In *Proceedings of the Eleventh International Conference on Machine Learning.* Morgan Kaufmann, San Mateo, CA, 1994.
12. N. Lavrač and S. Džeroski. *Inductive Logic Programming: Techniques and Applications.* Ellis Horwood (Simon and Schuster), 1994.
13. E. McCreath and A. Sharma. Extraction of meta-knowledge to restrict the hypothesis space for ILP systems. In X. Yao, editor, *Proceedings of the Eighth Australian Joint Conference on Artificial Intelligence*, pages 75–82. World Scientific, November 1995.
14. E. McCreath and A. Sharma. ILP with noise and fixed example size: A Bayesian approach. In *Fifteenth International Joint Conference on Artificial Intelligence*, volume 2, pages 1310–1315, 1997.
15. C.J. Merz and P.M. Murphy. UCI repository of machine learning databases, 1996.
16. K. Morik, S. Wrobel, J-U. Kietz, and W. Emde. *Knowledge Acquisition and Machine Learning: Theory, Methods and Applications.* Academic Press, 1993.
17. S. Muggleton, editor. *Inductive Logic Programming.* Academic Press, 1992.
18. S. Muggleton and L. De Raedt. Inductive Logic Programming: Theory and Methods. *Journal of Logic Programming*, 19(20):629–679, 1994.
19. S. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory, Tokyo*, pages 368–381. Ohmsa Publishers, 1990. Reprinted by Ohmsa Springer-Verlag.
20. S. Muggleton, A. Srinivasan, and M. Bain. Compression, significance and accuracy. In *Proceedings of the Ninth International Conference on Machine Learning*, pages 338–347. Morgan Kaufmann, San Mateo, CA, 1992.
21. Stephen Muggleton. Inverse entailment and progol. *New Generation Computing Journal*, 13, May 1995.
22. S.H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming.* LNAI Tutorial 1228. Springer-Verlag, 1997.
23. J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, 1990.
24. Elaine Rich. *Artificial Intelligence.* McGraw Hill, 1983.

25. Toby Walsh. Depth-boundid discrepancy search. In *Fifteenth International Joint Conference on Artificial Intelligence*, volume 2, pages 1388–1393, 1997.
26. C. S. Wetherall. Probabilistic languages a review and some open questions. *ACM Computing Surveys*, 12(4):361–377, 1980.
27. J Zelle, R Mooney, and J Konvisser. Combining top-down and bottom-up techniques in iductive logic programming. In *Proceedings of the Eleventh International Conference on Machine Learning*, pages 343–351, 1994.