Induction in First Order Logic from Noisy Training Examples and Fixed Example Set Sizes

Eric McCreath

A Dissertation submitted to the School of Computer Science

and Engineering of The University of New South Wales

in fulfillment of the requirements for the degree of

Doctor of Philosophy

October, 1999

Abstract

This dissertation investigates the field of inductive logic programming (ILP) and in so doing an ILP system, Lime, is designed and developed. Lime addresses the problem of noisy training examples; learning from only positive, only negative, or both positive and negative examples; efficiently biasing and searching the hypothesis space; and handling recursion efficiently and effectively.

The Q-heuristic is introduced to address the problem of learning with both noisy training examples and fixed numbers of positive and negative training examples. This heuristics is based on Bayes rule. Both a justification of its derivation and a description of the context in which it is appropriately applied are given. Because of the general nature of this heuristic its application is not restricted to ILP.

Instead of employing a greedy covering approach to constructing clauses, Lime employs the Q-heuristic to evaluate entire logic programs as hypotheses. To tame the inevitable explosion in the search space, 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 using the Q-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 are provided to illustrate how Lime achieves its design goals of better noise handling, learning from a fixed set of examples (e.g., from only positive data), and of learning recursive logic programs. These results compare the performance of Lime with other leading ILP systems like Foil and Progol in a variety of domains. Empirical results with a boosted version of Lime are also reported.

The dissertation also makes an attempt to characterise the learning ability of the Q-heuristic. To this end, some preliminary results are presented to show that the Q-heuristic stochastically learns in the limit some restricted classes of concepts. This includes cases where the training set may contain: noise, only positive examples, or only negative examples.

Declaration

I hereby declare that this submission is my own work and to the best of my knowledge it contains no material previously published or written by another person, nor material which to a substantial extent has been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis.

I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and the conception or in style, presentation and linguistic expression is acknowledged.

....

Acknowledgements

I am greatly indebted to my supervisor, Arun Sharma. His insight and opinions have helped reshape my approach to ILP. He has also been good in pushing me along with my research. Arun's guidance in preparing publications was really valuable, as was his guidance and assistance in generating this text.

This research has been supported by an Australian Postgraduate Award and I am grateful for their financial support.

I would also like to thank Ross Quinlan for allowing me to attend his excellent and succinct subject on ILP during the first year of the doctoral program. Also, I would like to thank him for his valuable feedback over the years.

I acknowledge and thank Claude Sammut for his encouragement and his pragmatic insight into machine learning.

UCI Machine Learning Repository proved to be a very useful resource as much of the raw data used in the empirical studies was obtained from there. The repository of data provided by the Oxford group was also useful.

Preliminary versions of parts of this work have had the benefit of being reviewed for earlier conferences [68, 69]. I am grateful to several anonymous reviewers for their comments.

I would like to thank Michael Bain for his helpful comments during his time at UNSW and for bringing Muggleton's *"Learning from positive data"* paper [87] to my attention.

Thanks to Mark Reid, Eric Martin, George Katz, Phil Preston, and Margaret McCreath for proof reading aspects of this dissertation.

To my most excellent wife Margaret,
and to my parents Alex and Erica.

Contents

Acknowledgements **2**

List of Figures **8**

List of Tables **10**

1 Introduction **12**

1.1 Overview of LIME 13

1.1.1 Limitations 15

1.2 Related Work 16

1.3 Guide Map 16

2 Logical Preliminaries & Notation **18**

2.1 Logic Programming 18

2.2 Sets and Logarithms 19

2.3 Probability 19

3 Background **21**

3.1 The Philosophy of Induction 21

3.2 A Computational Approach to Induction 22

3.3 Inductive Logic Programming 22

3.4 Subsumption and Generalization 23

3.4.1 Reynolds — algebraic structure of formulas 24

3.4.2 Plotkin — notes on generalization 25

3.5 Shapiro — The Model Inference System 25

3.6 Inverse Resolution 29

3.7 Quinlan — The FOIL System 31

3.8 Muggleton — Relative Least General Generalization 32

3.9 Inverse Implication 35

3.10 Greedy Covering Algorithm 35

3.11 Dimensions of ILP systems 35

3.11.1 Top-down and bottom-up 35

3.11.2 Batch learning and incremental learning 36

3.11.3 Single and multiple predicate learning 36

3.11.4 Interactive and noninteractive 37

3.11.5 Language bias . . . 37
3.11.6 Search bias . . . 39
3.11.7 Validation bias . . . 39
3.11.8 Imperfect data . . . 39
3.11.9 Example driven and example set driven . . . 40
3.11.10 Predicate invention . . . 41
3.12 Implementations . . . 41

4 Noise model and the Bayesian Heuristic **49**
4.1 Framework . . . 49
4.2 Q heuristic . . . 51
4.3 Theory . . . 52
4.4 Discussion . . . 54

5 Lime **56**
5.1 Hypothesis Space . . . 56
5.2 Simple Clauses . . . 57
5.3 Preprocessing the Background Knowledge . . . 61
5.3.1 Extracting type . . . 62
5.3.2 Extracting mode information . . . 62
5.3.3 Removing redundancy . . . 62
5.3.4 Improving indexing . . . 63
5.4 Simple Clause Table Construction . . . 63
5.4.1 Algorithm for simple clause coverage table and variable binding table . . . 66
5.5 Clause Table Construction . . . 66
5.5.1 Other approaches . . . 72
5.5.2 Efficiency . . . 72
5.6 Inducing the Final Hypothesis . . . 73
5.6.1 Recursive logic programs . . . 75
5.7 Prior Probability . . . 77
5.8 θ Estimation . . . 79

6 Empirical Results **80**
6.1 Simple Tests . . . 80
6.1.1 Boolean functions . . . 80
6.1.2 Family relationships . . . 80
6.2 Recursive Logic Programs . . . 81
6.2.1 Common recursive relations . . . 81
6.2.2 Example logic programs form Bratko's textbook . . . 83
6.2.3 Quick sort . . . 83
6.3 Noise . . . 85
6.3.1 Plus two . . . 85
6.3.2 Addition . . . 86
6.3.3 KRK domain . . . 87

6.4 Randomly Generated Domain . . . 87
6.4.1 Number of clauses . . . 88
6.4.2 Number of literals in each clause . . . 88
6.4.3 Number of predicates in the background knowledge . . . 89
6.4.4 Noise in the training set . . . 89
6.5 Learning from Positive Examples and from Negative Examples . . . 91
6.5.1 Plus two . . . 91
6.5.2 Addition . . . 92
6.6 Real World Data . . . 94
6.6.1 Mesh resolution . . . 94
6.6.2 Student loan . . . 94
6.6.3 Tic-Tac-Toe . . . 95
6.6.4 Wisconsin breast cancer . . . 97
6.6.5 Votes . . . 97
6.7 Past Tense of English Verbs . . . 98
6.7.1 Document understanding . . . 99
6.7.2 Protein secondary structure . . . 102
6.7.3 Mutagenesis . . . 102

7 Probabilistic identification results **105**
7.1 Probabilistic Identification in the Limit . . . 105
7.2 Probabilistic Identification with the Q-heuristic . . . 112
7.2.1 A finite concept class . . . 112
7.2.2 A restricted infinite concept class . . . 116
7.3 Example Concept Classes . . . 122
7.4 A Modification of the Q-heuristic and the Learning Algorithm . . . 130

8 Boosting **133**
8.1 Introduction . . . 133
8.2 Description . . . 134
8.3 Boosting and ILP . . . 135
8.4 Lime . . . 136
8.5 Noise . . . 138
8.6 Empirical Results . . . 140
8.6.1 Student Loan . . . 141
8.6.2 Tic-Tac-Toe . . . 142
8.6.3 Wisconsin breast cancer . . . 143
8.6.4 Protein . . . 144
8.6.5 Votes . . . 144
8.7 Discussion . . . 145

9 Conclusion **146**
9.1 Summary . . . 146
9.1.1 Empirical evaluation of the Q-heuristic: LIME system . . . 146
9.1.2 Theoretical evaluation of the Q-heuristic: learnability results . . . 147
9.2 Future Work . . . 147

Bibliography **149**

A Statistics **158**
A.1 Estimating the mean and standard error . . . 158
A.2 Comparing the average accuracy or error of two learners . . . 159

B Lemmas for stochastic limit results **160**

C Meta-Knowledge **177**
C.1 Introduction . . . 177
C.2 An Illustration of the Approach . . . 179
C.2.1 Type . . . 180
C.2.2 Functional dependency . . . 181
C.2.3 Symmetry . . . 182
C.3 Experimental Results . . . 183
C.3.1 Examples from Bratko's book . . . 184
C.3.2 Examples that came with FOIL . . . 185
C.4 Theoretical Properties of the Algorithms . . . 186
C.5 Discussion . . . 186

List of Figures

1.1 The partitioning of LIME's stages. 14

3.1 A section of Reynolds atomic formula lattice 25
3.2 An overview of LINUS' approach 44

4.1 Model of positive example generation 50
4.2 Model of negative example generation 50

5.1 The partitioning of LIME's stages. 57
5.2 Directed graph corresponding to the example clause 58
5.3 Directed graph corresponding to the over twelve clauses 58
5.4 Tables constructed in the simple clause stage for *add* 64
5.5 The search tree where gain_list_length = 100 68
5.6 Example of estimating coverage of a individual recursive clause 76

6.1 A family tree 81
6.2 Predictive Error vs Noise for *plus2* 86
6.3 Error vs Noise for *add* 87
6.4 The predictive error as noise is introduced into the training examples in the KRK domain. 88
6.5 The predictive error vs number of clauses in the randomly generated domain. . . . 89
6.6 The predictive error vs number of literals in each of the 3 clauses in the randomly generated domain. 90
6.7 The predictive error vs number of predicates in the background knowledge in the randomly generated domain. 90
6.8 The predictive error vs noise introduced into the training examples in the randomly generated domain. 91
6.9 Error vs i (i = number of positive examples & 8-i = number of negative examples) for the *plus2* and *notplus2* logic programs. (e.g., At $i = 6$ LIME is provided 6 positive examples and 2 negative. This number of examples is sufficient(insufficient) for LIME to learn the *plus2*(*notplus2*) relation.) 92

6.10 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 trials are conduced with different levels of noise used in generating the training examples. 93
6.11 LIME inducing *addition* from only positive examples. The left graph shows error vs number of training examples. The right graph shows the number of times a correctly induced hypothesis is generated out of 10 trials vs number of training examples. 93
6.12 Board position with crosses winning and its representation in first order logic. . . 96
6.13 Accuracy verse number of training examples for the verb past tense data set. . . . 99
6.14 Number of clauses in induced hypothesis verse number of training examples for the verb past tense data set. 100
6.15 Page layout of the letter labelled *x1*. 101
6.16 Comparison, between LIME and FOCL, of test result for each of the logical components considered. 102

7.1 The cover of h_t and h_δ. 113

8.1 Example illustrating the boosting process. 136
8.2 Overview of stages of boosted LIME. 137
8.3 Comparison of LIME with and without boosting as noise is introduced using the the king-rook-king example. 141

B.1 Division of region into 5 separate cases. 161
B.2 e_p is shown to be at or above the "wedge". 165
B.3 e_p is shown to be at or above a "wedge" when $\delta_a = 0$ or $\delta_b = 0$. 169

C.1 (a) ILP System, (b) ILP System with property generator 178

List of Tables

5.1 Grammar use by LIME for calculating prior of hypothesis. 78
5.2 Stochastic expectation matrix M for the logic program probabilistic context-free grammar. 78

6.1 Learning simple boolean logic programs 80
6.2 The positive examples and part of the background knowledge used when inducing the *parent* relation. 82
6.3 Some recursive logic programs 82
6.4 Bratko's recursive logic programs 84
6.5 Empirical results of LIME, FOIL, and PROGOL for recursive logic programs from Bratko's textbook. 85
6.6 Number and percentage of correctly classified mesh sizes by FOIL, MFOIL, GOLEM, and LIME for each of the test structures. Note, the result for FOIL, MFOIL, and GOLEM are given in [61]. 95
6.7 Results for LIME, FOIL, and PROGOL on the student loan data set. 95
6.8 Results reported for the tic-tac-toe domain in [4] 96
6.9 LIME's results for the tic-tac-toe domain. 96
6.10 Result for breast cancer domain. 97
6.11 Result for breast cancer domain with 10 10-fold cross-validations 97
6.12 Result in votes domain using 10 10-fold cross validation. 98
6.13 A hypothesis induced in the past tense domain. 98
6.14 A portion of the background knowledge used in the document understanding domain.100
6.15 Number of molecules in each partition for the mutagenesis data set. 103
6.16 Results for the mutagenicity data set. 103

8.1 Logic for the king-rook-king illegal move. 140
8.2 Results for LIME with and without boosting in the student loan domain. 141
8.3 Results reported for the tic-tac-toe domain in [4] 142
8.4 Results for the tic-tac-toe domain with and without boosting. 142
8.5 Results for breast cancer domain. 143
8.6 Results for breast cancer domain with ten 10-fold cross-validations 144
8.7 Results of protein domain with and without boosting. 144
8.8 Results of votes domain with and without boosting using 10 10-fold cross validation.145
8.9 Results of votes domain with and without boosting using 15 10-fold cross validation.145

C.1 ILP Systems using meta knowledge 178
C.2 Raw information for learning *merge* 180
C.3 Processing Times — Examples from Bratko's PROLOG book. 184
C.4 Processing Times — FOIL's given examples. 185

Chapter 1

Introduction

Inductive Logic Programming (ILP) is the study of learning systems that induce logic programs as hypotheses from ground facts and background knowledge. Chapter 3 introduces ILP formally and contains a literature survey of the area. In the present chapter, we give an overview of the dissertation and its main contributions.

Current ILP systems are limited in their ability to handle noise in the training data, as they employ a greedy covering approach to constructing the hypothesis one clause at a time. This approach also causes difficulty in learning recursive predicates. Additionally, many current systems have an implicit expectation that the cardinality of the positive and negative examples reflect the "proportion" of the concept to the instance space. This dissertation addresses these problems and proposes possible solutions.

The three main contributions of this dissertation are:

- development of the Q-heuristic for assessing the quality of a hypothesis in a Bayesian framework that allows modelling of noise and fixed training example size,
- design and development of the ILP system Lime that employs the Q-heuristic, and
- a preliminary theoretical analysis of the the Q-heuristic providing some initial stochastic identification in the limit results.

A framework for learning from noisy data and fixed training example size is presented. A Bayesian heuristic for finding the most probable hypothesis in this general framework is derived. This approach evaluates a hypothesis as a whole rather than one clause at a time. The heuristic, which has nice theoretical properties, is incorporated in an ILP system, Lime. Experimental results show that Lime handles noise better than Foil and Progol. Lime is able to learn recursive definitions from noisy data on which other systems do not perform well. Lime is also capable of learning from only positive data and from only negative data. The heuristic is also examined in terms of its behaviour as more and more examples are presented to it. The heuristic is shown to stochastically learn in the limit some restricted classes of concepts.

In the rest of this chapter we give a detailed overview of the Lime system and discuss the rationale behind its design, we note some of its limitations and describe related work. We end the chapter with a guide map for rest of the dissertation.

1.1 Overview of LIME

LIME is an ILP system that induces logic programs as hypotheses from ground facts. Unlike many systems (e.g., Quinlan's FOIL [104], Muggleton and Feng's GOLEM [90], and Muggleton's PROGOL [94]) that employ a greedy covering approach to constructing the hypothesis one clause at a time, LIME employs a Bayesian framework that evaluates entire logic programs as candidate hypotheses. The system LIME, developed in this framework, has the following features:

- better noise handling,
- 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[1], and
- improved ability to learn predicates with recursive definitions.

Since the Bayesian heuristic employed in LIME requires evaluation of entire logic programs as hypotheses, the search space is naturally huge. This dissertation 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. The coverage of the new 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.

Most ILP systems like GOLEM [90] and FOIL [104] employ a greedy covering heuristic to build hypotheses. They try to find the clause that covers the maximum number of positive examples without covering any, or few, negative examples. A new set of positive examples is created by removing the covered positive examples and the process is repeated with this new set of positive examples and the negative examples until there are no uncovered positive examples left. While this approach has led to efficient learning in many applications, there are situations in which it fails to perform well. The greedy covering algorithm is shown in Algorithm 5 on page 36.

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

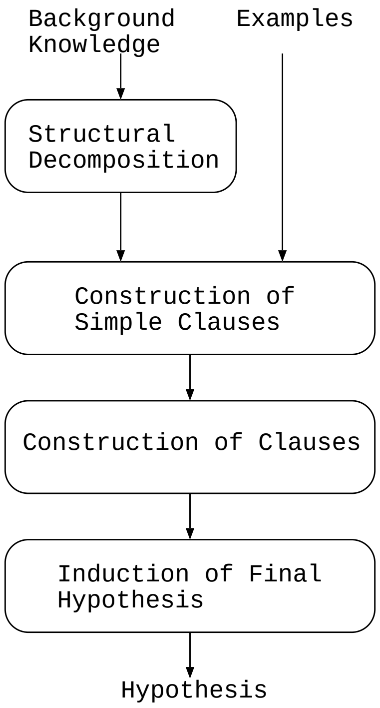


Figure 1.1: The partitioning of LIME's stages.

Consider the problem of noise handling. GOLEM [90] has a rudimentary noise handling facility as each clause is allowed to cover a fixed number of negative examples in addition to as many positive examples as possible. This approach is inflexible as it requires adjusting the noise parameter with changes in sample size and noise level. Inflexibility aside, handling of noise at the clause level, in many cases, appears to result in a poor overall hypothesis, mainly because of over-fitting. FOIL [104] employs an MDL (Minimum Description Length) [109]/MML (Minimum Message Length) [130] like approach to noise handling. While this is better than an *ad hoc* approach, its greedy covering strategy of building the hypothesis one clause at a time can lead to over-fitting in the presence of noise (for example, see Figure 6.2 that describes FOIL's performance on a simple predicate *plus2*).[2] A single clause only yields information about over-generalization errors. Such information about data not covered by the hypothesis requires looking at the complete hypothesis.

It may also be argued that Bayesian approaches like MDL/MML are not very appropriate to greedy covering methods, because they are best suited to estimating the posterior probability of the whole hypothesis and not of individual clauses in the hypothesis.

Another problem with current ILP systems arises while learning predicates that require recursive definitions. In these situations, for successful learning to take place, a system has to be provided with a complete initial sequence of the data, including the base case. However, if the data is sparse, it becomes very difficult for these systems to learn hypotheses with recursive clauses. Again, an approach that evaluates complete hypotheses instead of individual clauses does a better

[2] mFOIL [62] has an improved noise handling capability, but it still suffers from the greedy covering approach. Noise handling in LINUS [60] is more about taking advantage of noise handling techniques from attribute value learning.

job of overcoming such deficiencies in the data.

Apart from the above mentioned difficulties, if an ILP system simply attempts to maximize the number of correctly classified examples, then it has 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 learning situations this assumption is unjustified. Usually the cardinality 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. Moreover, many systems require a minimum number of positive and negative examples. For example, Progol requires positive examples to seed the clauses that eventually form the induced hypothesis whereas Foil requires negative examples, either given explicitly or generated implicitly via the closed world assumption.

Motivated by the above problems with existing ILP systems, the present thesis introduces a general framework for noise handling in learning systems, and derives a Bayesian heuristic for inducing a hypothesis with the maximum posterior probability. The framework assumes that the number of examples is fixed independently of the concept (and certainly not representative of the actual proportion of the concept to the instance space). Hence, the heuristic can be employed to learn from only positive data and from only negative data in addition to the usual combination of positive and negative data. An interesting theoretical result about the framework is that it formalizes the intuitive importance of an extra positive example over an extra negative example for concepts that are "small" with respect to the instance space. A similar result for the importance of negative examples over positive examples holds for concepts that are "large" with respect to the instance space. Empirical evidence in keeping with these theoretical results are presented.

The heuristic is adapted for ILP and implemented in the Lime system which is discussed in Chapter 5. The experimental results obtained by Lime are then given in Chapter 6 showing the effectiveness of the Q heuristic. It should be noted that another system that looks at complete hypotheses instead of single clauses is Tracy [12].

When induction is undertaken in a probabilistic setting two subtle distinctions arise in the objective of the learning process. First, prediction, where the hypothesis generated is expected to maximize the predictive accuracy on unseen examples. This objective may be most appropriate when the induced hypothesis will be later used to classify unseen data. However, there are many practical difficulties in applying this approach. Second, concept discovery, where the induced hypothesis is the most likely hypothesis to represent the target concept. This objective may be most appropriate if the aim is to discover the concept that underlies the data given to the learner. Concept discovery is the setting assumed throughout this thesis. Although, in general, concept discovery will not yield the best predictor, it will still result in a good predictor.

1.1.1 Limitations

An important limitation of this work is the restrictions on the hypothesis space. Clauses are determinate and do not contain function symbols. Also, the background knowledge is required to be extensional. Although, this limits Lime's capability, it does not limit the applicability of the heuristic, as the Q heuristic is independent of the hypothesis space and the search bias. Another scope for improvement is extending Lime to handle floating point constants naturally. Some of these issues will be addressed in future work.

1.2 Related Work

We refer the reader for an overview of ILP to the book by Nienhuys-Cheng and de Wolf [95] or the article by Muggleton and De Raedt [89]. Other source books for ILP are Bergadano and Gunetti [13], Lavrač and Džeroski [61], and Muggleton [84]. 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 [37] describe the transformation of determinate ILP problems into propositional form. The systems Linus and Dinus by Lavrač and Džeroski [61] employ attribute value learners after transforming restricted versions of the ILP problem.

Kietz and Lübe [49] introduced the notion of k-local clauses which is somewhat similar to simple clauses. They divided a clause into determinate and nondeterminate parts and further subdivided the nondeterminate part into k-local 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 [11]. During the preprocessing phase of the background knowledge, Lime automatically extracts type and mode information. Similar issues are addressed by Morik et al [81] in their system Mobal.

mFoil [34, 35] extends Foil by incorporating a more robust noise-handling mechanisms. Lime also address the problem of noise, although, mFoil deals with the problem of noise within the greedy covering approach using either a Laplace error estimate or the m-estimate to direct the construction of clauses. Also, a significance testing approach is used as a stopping criterion. Whereas, Lime addresses the problem of noise by considering entire hypotheses.

Latex [127] and Muse2 [135] are theory revision systems and are related to Lime in the approach used to deal with noisy datasets. These systems use the MDL/MML principle for revising and learning.

If noise is removed and only positive data is considered then the Q heuristic presented in Chapter 4 simplifies to Muggleton's heuristic presented in *"Learning from positive data"* [87].

The theoretical learnability results of Chapter 7 are on the same line as those of Laird [56].

1.3 Guide Map

The rest of the dissertation is organised as follows. Chapter 2 introduces the notation of logic programming and inductive logic programming, and also describes aspects of probability theory that relate to this research. The following chapter presents a background to ILP, considering a variety of approaches and systems. In Chapter 4, noise is investigated and a framework for induction is presented. From this framework the Q heuristic is formulated. The Lime system is motivated and described in detail in Chapter 5. A variety of empirical results are presented in

Chapter 6. The stochastic limit results for the Q-heuristic are presented in Chapter 7. Boosting is studied in relation to LIME in Chapter 8. Finally, in Chapter 9 this dissertation is concluded.

The dissertation also contains three appendices. Appendix A briefly states the statistical techniques used for the empirical studies of Chapters 6 and 8. The next appendix contains some technical detail for identification in the limit results presented in Chapter 7. Appendix C describes an approach of extracting meta-knowledge for ILP systems in general. This approach is also used in LIME, enhancing the overall performance of the inductive process.

Chapter 2

Logical Preliminaries & Notation

Before proceeding, the notation of logic programming and inductive logic programming is introduced in this chapter. This notation is based on Lloyd [65] and the notion of induction is based on Shapiro [122] and Plotkin [99]. This section also clarifies the notation used throughout the paper. Some of the inductive logic programming notation is based on Lavrač and Džeroski [61]. Probability theory is also introduced, as it is central to the section in the thesis relating to the Q heuristic. The reader should refer to any introductory text, such as [111, 131, 7], for more information.

2.1 Logic Programming

A first-order language is denoted $\mathcal{L}$. A *term*, denoted usually by t with or without subscripts, is either: a variable, a constant, or an n-ary function applied to n terms. *Variables* are denoted by either V_i or A, B, C, D, E, F; *constants* are denoted by a, b, c, and *function symbols* by f, g, h. Variables and constants may be associated with a type, these are denoted with Greek letters τ, σ. Function symbols may also be associated with a vector mapping type $\tau_1 \times \tau_2 \times \cdots \times \tau_n \to \tau$, where τ is the type of the function, τu_i is the type of the i'th argument in the function. *Atoms* $p(t_1, t_2, \ldots, t_n)$ of arity n are constructed with a predicate symbol p, and terms $t_1, t_2, \ldots, t_n$. Predicates may be associated with a type vector $\tau_1 \times \tau_2 \times \cdots \times \tau_n$. A *literal* is either an atom or the negation of an atom, and is usually denoted with the letter L with or without subscripts. A *definite clause* $A \leftarrow B_1, B_2, \ldots, B_n$ is a set of literals consisting of exactly one positive literal. Definite clauses may be represented as a set of literals $\{A, \neg B_1, \neg B_2, \ldots, \neg B_n\}$. If $n = 0$ then the definite clause is a *fact*. A *formula* is either an atom or is obtained from repeated applications to formulas with the following operations: negation, disjunction, conjunction, universal quantifier, and existential quantifier. A *logic program* is a finite set of definite clauses. Note that, logic programs are a restricted class of formulas. Logic programs are denoted with the letters P or G. A logic program may have a type assignment associated with it, this type assignment consists of three mappings, firstly between constant symbols and their type, secondly between function symbols and their type mappings, and thirdly between predicates and their type vectors.

An *interpretation* attaches meaning to the symbols in a formula. If the formula is true in an interpretation then the interpretation is a model for the formula. A formula P is *satisfiable* if there exists an interpretation for P which is a model. A formula P is *valid* if all interpretations of P are models for P. $P \models G$ means $P \rightarrow G$ is valid. The *Herbrand Universe* $U_{\mathcal{L}}$ of a first-order language $\mathcal{L}$ is the set of all ground terms that can be formed in $\mathcal{L}$. A *Herbrand base* $B_{\mathcal{L}}$ is the set of all ground atoms that can be formed in $\mathcal{L}$. The *Herbrand interpretation* can be identified by a subset of $B_{\mathcal{L}}$, where elements in the subset are true in the Herbrand interpretation. A *Herbrand model* for P, denoted $\mathcal{M}(P)$, is a Herbrand interpretation which is a model for P.

2.2 Sets and Logarithms

A *set* S is a collection of objects called *elements*. The set consisting of the naturals $\{1, 2, 3, \dots\}$ is denoted $\mathbb{N}$ and $\mathbb{R}$ denotes the real numbers. $x \in S$ denotes that x is an element of S and $x \notin S$ denotes x is not an element of S. The *empty set* is denoted $\emptyset$. The number of elements in a set is denoted $|S|$. $\subseteq$ and $\subset$ denote subset and proper subset respectively. The *power set* of set A, 2^A, is the set of all subsets of A. The standard set operations of complement, union, intersection, and subtraction are denoted $\overline{A}$, $\cup$, $\cap$, and $-$ respectively. The symmetric difference, denoted $A \Delta B$, is defined $\{x | (x \in A \wedge x \notin B) \vee (x \notin A \wedge x \in B)\}$.

The natural logarithm is denoted ln, so $\ln(x) = \log_e(x)$. Also, lg denotes the logarithm with base 2, that is $\lg(x) = \log_2(x)$.

2.3 Probability

Generally probability measures, denoted with P, are interpreted as a frequency, these are known as objective probability. Let "X is *heads*" denote the event that heads comes up when a coin is flipped. So, we interpret $P(X \text{ is } \textit{heads}) = 0.5$ as 50% of the time in repeated experiments a coin will come out heads. However, we may also interpret a probability measure as a "degree of belief" these are known as subjective probabilities. This interpretation is at the heart of Bayesian techniques and prior distributions.

Suppose we are given a set S, a class $\mathcal{F}$ of subsets of S (which is a field), and a probability measure P on $\mathcal{F}$. Let $A \in \mathcal{F}$ and $B \in \mathcal{F}$, then we have the following properties:

1. $0 \leq P(A) \leq 1$.
2. $P(S) = 1$, $P(\emptyset) = 0$.
3. $P(\overline{A}) = 1 - P(A)$.
4. $P(A \cup B) = P(A) + P(B) - P(A \cap B)$.
5. $P(A \cup B) \leq P(A) + P(B)$.
6. If $\{E_1, E_2, \dots, E_n\}$ partitions S where $E_i \in \mathcal{F}$ $(i = 1, 2, \dots, n)$ then:

$$\sum_{i=1}^{n} P(E_i) = 1$$

A *conditional probability* of A given B, denoted $P(A|B)$, is defined as $\frac{P(A \cap B)}{P(B)}$ where $P(B) \neq 0$. Note, $A|B$ is not a set construct.

Events A and B are *independent* if $P(A \cup B) = P(A)P(B)$.

A *discrete random variable* X is a mapping from S into $R \subset \mathbb{R}$, where R is the range and is countable. The *expected value or mean* of a random variable is $E(X) = \sum_{x \in R} xP(X = x)$. A *discrete uniform random variable* is a random variable X where $n = |R|$ and for all $x \in R$, $P(X = x) = \frac{1}{n}$. In the *Bernoulli random variable* R is taken to be $\{0, 1\}$ and $P(X = 1) = p$ and $P(X = 0) = 1 - p$ where $0 \leq p \leq 1$.

Chapter 3

Background

This chapter provides a background to ILP, reviewing important developments in the field and giving an overview of most implemented systems.

3.1 The Philosophy of Induction

Induction refers to any kind of inference in which we move from a finite set of observations about an "object" or a "concept" to a conclusion that is a general description of the object or the concept. This is in contrast to *deduction* where the premises provide definite reasons for the conclusion, making any knowledge gained in the conclusion both certain and justifiable. Clearly induction, unlike deduction, makes a claim on knowledge that extends beyond the observed facts. In making this leap, induction gains its utility as an inference mechanism, yet loses the certainty and justification of deduction. Because induction draws a general conclusion about unseen facts, the conclusion gained may be used for prediction. This, in some sense, is the basis of the empirical method.

However, as noted by Hume [46], the inductive conclusion is not justifiable, as it goes beyond the set of observations. Several attempts have been made to overcome this problem; some notable ones are briefly discussed below.

- The inductive argument for induction posits that since induction has worked well in the past, therefore, by induction, it will always work (e.g., see Black [16, chap 11]). However, this is essentially a circular argument.

- In Karl Popper's [102] view, science does not rest on induction, rather, it is a process of disproving conjectures—a completely deductive process. According to Popper, a currently held theory is not proven true, rather, it just has not been shown to be false. So the value of a theory lies in its ability to be falsified. However, this does not change the uncertainty and unjustifiablity of the current theory.

- Another approach is to view inductive conclusion as "probable" rather than "certain". This attaches subjective probabilities to a theory which requires *a prior* information. This is also problematic as the probabilities attached to a theory are similarly not justifiable.

The lack of justification associated with induction as an inference mechanism is a fundamental problem (see Salmon [113] for an extensive discussion of the issues involved).

In the 1950's and 1960's, various philosophers of science, such as Popper, became less enamoured of induction. The idea of induction, however, was embraced by the rapidly expanding field of *Artificial Intelligence*, resulting in development of numerous practical learning systems. Also, researchers in theoretical computer science became interested in the mathematical foundations of induction resulting in the field of *Computational Learning Theory*.

Additional discussion of philosophical issues pertaining to induction can be found in any introductory Philosophy text such as Grayling [43].

3.2 A Computational Approach to Induction

The description of induction in the previous section is very general. A more detailed model is required to make rigorous arguments about induction. In this thesis we take a computational approach to induction outlined below.

Consider a universe of objects, U. Elements of U are referred to as *instances*. A *concept* is any subset of U. Let C be a concept. Then an element of C is a *positive example* of C and an element of $U - C$ is a *negative example* of C.

A concept may be infinite in size. An *intensional description* of a concept is a finite "description" that characterizes the concept. An example of an intensional description is a decision procedure for the concept.

Now let us assume that nature chooses an underlying reality in the form of a concept, also referred to as the *target concept*. A *learner* is a computational agent that attempts to find an intensional description of the target concept. To assist the learner in its task, nature provides the learner with *data* about the target concept. This data is usually a finite set of labelled instances from U; the label being 1 (0) if the instance is a positive (a negative) example of the target concept.

In the computational setting of induction, the learner has access to a hypothesis space which is essentially a sequence of intensional descriptions. It is useful to assume that this space of hypotheses contains at least one hypothesis that is a description of the target concept.

The task of the learner is to find a hypothesis for the target concept based on the data presented to it about the concept.

The conditions under which a learner is said to learn a target concept is referred to as the *criterion of success*.

In the next section, we describe Inductive Logic Programming—a particular computational setting for induction. The intensional descriptions in this setting are logic programs.

3.3 Inductive Logic Programming

Inductive Logic Programming (ILP) is concerned with learning logic programs as hypothesis from examples. ILP can be viewed in many different ways:

- machine learning where the hypothesis language is a subclass of first-order logic,

- logic program synthesis,
- system explanation and prediction, or
- data compression.

The idea of learning with predicate logic as a description language goes back to Banerji [9] in 1964. Plotkin [99] in 1971 and Shapiro [122] and Sammut [115] in 1981 laid important foundations for ILP. In and mid-1970's Vere [128] wrote programs based on Plotkin's and Reynold's work. Also, in the late 1970's Michalski [73] developed first order learning systems. Another early system was CONFUCIUS developed by Cohen [27]. However, the field has seen a resurgence in the 1990's and has been named *inductive logic programming* by Muggleton [83, 84]. Muggleton described it as the intersection of machine learning and logic programming. This is appropriate as the field utilizes these two well established areas. ILP, which is a particular setting for computational induction, uses first-order logic as the language of description and inference. The learner is also given background knowledge, which is a finite set of clauses, to aid and bias the induction. For a more complete review of the origins of ILP we refer the reader to Sammut's [114] prehistoric tale.

The *normal problem* setting for ILP, sometimes referred to as the *explanatory* setting, is: Given background knowledge B, positive examples E^+, and negative examples E^-, the learner finds a hypothesis that together with the background knowledge explains all the positive (complete) and none of the negative (consistent) examples. The examples are generally restricted to ground atoms, although, this is not essential. The reader should note that this is a very general description of the problem, there may not always be a solution. We formalize some technical notions next.

Definition 3.3.1 *If* $P = \{C_1, C_2, \dots\}$ *is a set of clauses then* $\overline{P}$ *denotes* $\{\neg C_1, \neg C_2, \dots\}$.

Definition 3.3.2 (Consistent) *A logic program* P *is consistent with respect to a set of examples* E^- *iff* $P \cup \overline{E^-}$ *is satisfiable.*

Definition 3.3.3 (Complete) *A logic program* P *is complete with respect to a set of examples* E^+ *iff* $P \models E^+$.

Definition 3.3.4 (ILP: Normal problem setting) *Given a finite set of clauses* B *and sets of clauses* E^+ *and* E^-*, where* B *is the background knowledge,* E^+ *is the set of positive examples, and* E^- *is the set of negative examples, find a hypothesis* H*, if such a hypothesis exists, such that* $H \cup B$ *is complete with respect to* E^+ *and consistent with respect to* E^-.

There are a few difficulties in defining ILP in this way: first there is no allowance for noise, second there may be many solutions that will be both consistent and complete. Additionally, the hypothesis may be complete and consistent with respect to the examples given, but may not be a good predictor about the underlying concept on unseen examples.

3.4 Subsumption and Generalization

Gordon Plotkin [99, 98, 100] formulated an important theoretical foundation for inductive generalization, pioneering the then unnamed field of ILP. There had been some investigations into generalization in predicate calculus by Melter [70], who was able to find general axioms for group theory systems from particular examples. Also, Popplestone in [47] looked at the problem of finding a function that satisfies a given argument-value table: Popplestone was not concerned predominantly with generalization, rather focusing on techniques for generating hypotheses.

We first introduce Reynolds' [107] work as it is in essence a special case of Plotkin's ideas. Reynolds' paper is not directed at machine learning, rather it aims to build a theoretical basis for proof procedures in logic programming. However, it does present an important basis for ILP.

There is considerable overlap between the work of Plotkin and Reynolds. Interestingly enough both *Transformational System and the Algebraic Structure of Atomic Formulas* by Reynolds [107] and *A Note on Inductive Generalization* by Plotkin [98] appeared in the same collection. Moreover, they referred to each other's work, claiming independent discovery of the notion of least general generalization algorithm for atoms. Reynolds and Plotkin used different symbols for subsumption, with Reynolds using '$\geq$' and Plotkin using '$\leq$', creating some confusion. The convention used in this thesis is that of Reynolds: A *subsumes* B if there is a substitution θ such that $A\theta = B$, which is denoted $A \geq B$. We follow this approach as A is more general and hence may be viewed "greater" or "stronger" than B.

3.4.1 Reynolds — algebraic structure of formulas

There is a natural generalization ordering for formulas. That is, A is at least as general as B iff for any interpretation I that is a model for A, I is also a model for B. Consider the formula $p(X)$, any interpretation I must interpret $p(a), p(b), \cdots$ as being true. So, I must be a model for the formula $p(d)$, hence, $p(X)$ is at least as general as $p(d)$. By restricting the formulas to the set of atomic formulas, this generalization ordering becomes straight forward as A is at least as general as B iff there exists a substitution θ such that $A\theta = B$. This creates a partial ordering on the equivalence classes obtained under variable renaming.

To create a lattice structure of atomic formulas, Reynolds introduces a *universal formula* Λ and a *null formula* Ω. Then the greatest common instance of two formulas can be obtained from Robinson's unification algorithm [110]. If the formulas are not able to be unified then the greatest common instance will be Ω. Also, the least general generalization of two formulas may be calculated. Reynolds provides an algorithm for this generalization. This proves to be useful in induction, as, a generalization may be formed from positive examples and as it is the least general it will be best positioned to not prove any of the negative examples. For example if s is the successor function symbol then $plus\_two(0, s^2(0))$ and $plus\_two(s^2(0), s^4(0))$ are two instances of the *plus_two* relation[1]. If the least general generalization is derived from these examples a description of *plus_two* is induced.

$$\mathrm{lgg}(plus\_two(0, s^2(0)), plus\_two(s^2(0), s^4(0)) = plus\_two(X, s^2(X))$$

A section of this lattice can be seen in Figure 3.1. Clearly, single atomic formulas are very limited in the concepts they can express.

[1]The term $s^k(X)$ refers to $\overbrace{s(s(\cdots s(}^{k}X)\cdots))$.

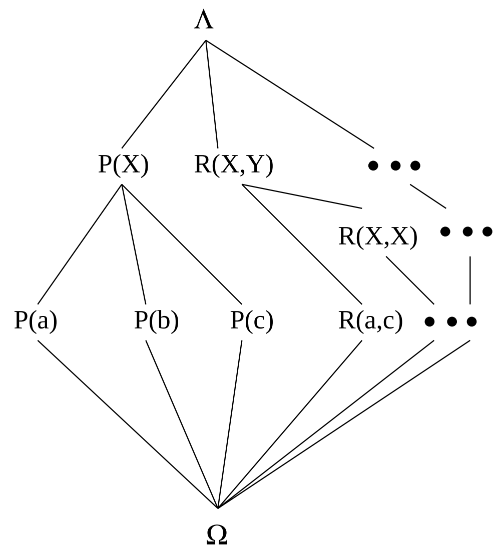


Figure 3.1: A section of Reynolds atomic formula lattice

3.4.2 Plotkin — notes on generalization

Plotkin's thesis [99, 100] is concerned with algorithms that generalize from experience. R.J.Popplestone suggested that as unification is useful for automated deduction then the dual of this operation, least general generalization, may be of value for automated induction. Plotkin developed an algorithm for finding a least general generalization of two clauses. Also, he was the first to establish that any finite set of clauses has a least general generalization under subsumption.

This research is a precursor to the relative least general generalization, which is central to Golem, Muggleton and Feng's [90] ILP system.

3.5 Shapiro — The Model Inference System

In the early 1980's Ehud Shapiro developed the Model Inference System(MIS) [122, 121]. This is an important system as it demonstrates the viability and utility of learning in first-order logic. Shapiro also showed how MIS identifies in the limit [42] a family of complex classes of concepts. Shapiro's work elegantly combines a practical implementation with theoretical analysis.

MIS is motivated from the philosophy of scientific discovery, in particular the Popperian methodology [102, 101]. The Popperian methodology is an iterative process where a hypothesis is induced which is consistent and complete with respect to the examples seen so far. If a counterexample is encountered then a new hypothesis, that is consistent and complete with respect to all the examples encountered so far, is induced. MIS also builds on the work of Reynolds [107] and Plotkin [99], extensively using the notion of refinement and refinement operators.

The model inference problem and the normal ILP setting are identical except for one key difference. The ILP setting assumes the learner receives a finite set of examples from which the learner must then induce a hypothesis, so, the learner may not choose which examples are provided. However, the model inference setting assumes that a complete enumeration of examples is available to the learner. This is equivalent to assuming that the learner can ask a question about the state of any instance in the observation language, thereby giving the learner an oracle for the concept. This is akin to the scientific method, as, a scientist may be uncertain about a portion of the concept, and hence, conduct an experiment to determine if it is included in the concept.

Initially, Shapiro defines the problem in general terms. Assuming the first-order language L is in clausal form with finitely many predicate and function symbols. Two subsets of this language are defined: the observational language L_o, and the hypothesis language L_h. It is assumed that $L_o \subset L_h \subset L$ and both sets are decidable.[2] A concept in this setting is a model M for L. The learner is required to find an L_o-complete axiomatization of M — a finite set of sentences $T \subset L_h$ that is true in M and is complete and consistent with L_o. The model inference problem is then described as given L, L_h, L_o and an enumeration of the concept, which is an enumeration of L_o labelled either to be part or not part of the concept, find a finite L_o-complete axiomatization for the concept. This enumeration effectively gives the learner an oracle with respect to L_o of the concept, as, if the learner wishes to ascertain if an instance is part of the concept it may simply wait until this instance appears in the enumeration. The reader should note that this setting differs from the normal setting of ILP where an oracle is not assumed, rather a fixed set of examples is provided to the learner.

The MIS framework is very general, as L_h is not restricted to Horn clauses and L_o is not restricted to ground facts. Now if L_o is not rich enough to distinguish the hypotheses generated from L_h, then clearly no learning system will be able to induce an L_o-complete axiomatization of the underlying model. Hence, attempting to construct such a learning system would be futile. Shapiro handled this problem by introducing the notion of the admissibility requirement on the expressiveness of L_o and L_h. The pair $\langle L_o, L_h \rangle$ is admissible if every false theory in L_h has a witness in L_o to its falsehood. This gives rise to the following taxonomy:

$\langle \{\text{atoms of } L\}, L \rangle$	is inadmissibile
$\langle \{\text{ground atoms of } L\}, \{\text{Horn sentences of } L\} \rangle$	is admissibile
$\langle \{\text{ground unit sentences of } L\}, L \rangle$	is admissibile

So when the examples presented to the learner are ground facts, it follows that the hypothesis language should be restricted to Horn sentences, because if this restriction is removed the learning problem becomes inadmissible. This forms the basic language restriction of most ILP systems.

The model inference problem is closely related to the problem of inductive inference where a learner is required to identify in the limit a program for a computable function from its graph. Gold [41, 42] showed that there is no learner that learns the entire class of computable functions. However, Gold was able to show that if the class of functions being learned was an enumerable class[3], then there exists a learner for the class. Such a learner is said to learn using the strategy "of identification by enumeration". Blum and Blum [17] showed that if there exists a computable upper bound on the complexity of computation of functions in a class, then the class is enumerable and hence learnable according to Gold's result. Shapiro mirrors this approach in the model inference domain. The class of h-easy models, where h is a computable function, consists of

[2] A hypothesis consists of a subset of L_h.
[3] A class is enumerable if there exist an enumeration of the entire class.

models M with a theory $T \subset L_h$ such that for all but finite $\alpha \in L_o$, if $T \vdash \alpha$, the proof procedure takes at most $h(\alpha)$ steps.

Shapiro constructs an identification by enumeration algorithm that identifies any h-easy model in the limit (shown in Algorithm 1). Although this approach will eventually induce the target

Input :
  $\{F_1, F_2, \ldots\}$ – An enumeration of examples.
Output :
  $\{H_1, H_2, \ldots\}$ – A list of hypotheses.
begin
  $S_{\text{false}} := \{\Box\}$
  $S_{\text{true}} := \{\}$
  $k := 0$
  $n := 0$
  repeat
    $n := n + 1$
    read the next fact $F_n = \langle \alpha, V \rangle$
    add α to S_V
    while $(\exists\ \alpha \in S_{\text{false}} \mid T_k \vdash_n \alpha) \vee (\exists\ \alpha \in S_{\text{true}} \mid T_k \not\vdash_{h(\alpha)} \alpha)$ **do**
      $k := k + 1$
    od
    output $H_n = T_k$
  forever
end

Algorithm 1: An Enumerative Model Inference Algorithm [122, Algorithm 1]

hypothesis, this is a very inefficient search of the hypothesis space. Shapiro addresses this problem by making use of the structure of first-order logic and incrementally changing the conjecture. The algorithm is shown in Algorithm 2.

Shapiro's model inference algorithm is often described as iteratively examining examples. If the current theory is too general, explaining a negative example, it uses the contradiction backtracing algorithm to remove the clause that makes it over general: and if the current theory does not explain a positive example then makes use of a refinement operator on previously removed clauses to introduce a new clause that will explain this positive example. In order to get a better insight, we review the notions of refinement operators and contradiction backtracing.

Shapiro assumes a structural complexity measure *size* over the sentences in L. The set of sentences of any *size* n is finite. This property naturally emerges in any structural complexity measure. Given the set of sentences S, $S(n)$ denotes the set of sentences from S with size less than or equal to n; i.e. $S(n) = \{q \in S | \text{size}(q) \leq n\}$.

Definition 3.5.1 (Refinement) *Let A and B be sentences drawn from the first-order language L. A is said to be a* refinement *of B if B logically implies A and size$(B) <$ size(A).*

Definition 3.5.2 (Refinement Operator) *A refinement operator ρ is a mapping from L to subsets of L which are their refinements. And for any $p \in L$ and any $n > 0$ the set of sentences $\rho(p)(n)$ is computable.*

Input :

$\{F_1, F_2, \dots\}$ – An enumeration of examples.

Output :

$\{H_1, H_2, \dots\}$ – A list of hypotheses.

begin

$S_{\text{false}} := \{\Box\}$
$S_{\text{true}} := \{\}$
$k := 0$
mark $\Box$ 'false'.
repeat
  $n := n + 1$
  read the next fact $F_n = \langle \alpha, V \rangle$, where $V \in \{\text{true}, \text{false}\}$
  add α to S_V
  while $(\exists\ \alpha \in S_{\text{false}} \mid L_\rho^m(k) \vdash_n \alpha)$ **do**
    apply the contradiction backtracing algorithm
    and mark the refuted hypothesis 'false'
  od
  while $(\exists\ \alpha \in S_{\text{true}} \mid L_\rho^m(k) \nvdash_{h(\alpha)} \alpha)$ **do**
    $k := k + 1$
  od
  output $H_n = L_\rho^m(k)$
forever

end

Algorithm 2: An Incremental Model Inference Algorithm [122, Algorithm 3]

This naturally introduces a partial order over L with a top element of $\Box$, where $\Box$ denotes the empty sentence. This partial ordering is denoted $\geq_\rho$ and we say $p \geq_\rho q$ if there is a finite ρ-chain from p to q, i.e., there exists a finite set of sentences $p = p_0, p_1, \ldots, p_n = q$ such that $p_{i+1} \in \rho(p_i)$ for $0 \leq i < n$. The set $\{p \in L | \Box \geq_\rho p\}$ is denoted ρ^*.

Definition 3.5.3 (Complete refinement operator) *Let $S \subset L$ and $\Box \in S$. A refinement operator ρ over L is complete for S, if $\rho^* = S$.*

A hypothesis which implies a false observational sentence is said to be too strong. If a hypothesis is too strong it must contain at least one false clause. The contradiction backtracing algorithm is used to detect such a false clause. This clause may be removed by weakening the hypothesis. Shapiro's algorithm works by considering the resolution tree which implies the false observational sentence and working back up the tree performing a limited number of crucial experiments to determine a false clause. These experiments are possible in MIS as the learner is provided an enumeration, which is effectively an oracle.

A refinement operator ρ gives rise to a refinement tree, where $\Box$ is the root of the tree. MIS partially grows this tree marking node elements that are 'false'. The current conjecture consists of clauses that have not been marked 'false' and are within the current size restriction. In effect when a positive example that should be implied by the hypothesis is not, the size restriction is increased until this sentence is implied. If a negative example that should not be implied, but is implied by the current conjecture is encountered, then the backtrace contradiction algorithm is employed and a false clause is found in the current hypothesis and marked appropriately in the refinement graph marking. This algorithm is similar to a candidate-elimination approach [75] where the marked refinement graph maintains a boundary on the eliminated hypotheses.

This algorithm is similar to the model directed, top-down approach of the Version Space [75] algorithm which converges by finding clauses that 'match' the data.

Shapiro shows that Algorithm 2 will identify in the limit any h-easy model. The system is implemented with a variety of refinement operators demonstrating MIS' effectiveness on several different problems. One draw back with this approach is that the conjectures induced generally contain many redundant clauses, although, they will not affect the model implied, they result in larger and cluttered hypotheses.

3.6 Inverse Resolution

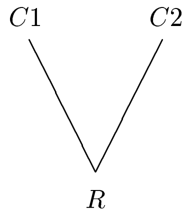
Inverse resolution is a tool for induction and was first introduced by Sammut and Banerji [115, 116] in the MARVIN learning system. It was later used in Muggleton's [82] DUCE system. A more complete examination of the process is presented by Muggleton and Buntine [88]. As the name suggests inverse resolution is inverting the deductive proof step of resolution. It is an important technique and gives rise to many ILP techniques.

Resolution [110] is a proof procedure based on the following rule: $\{(\phi \vee \psi), (\neg\phi \vee \chi)\} \models (\psi \vee \chi)$. Clearly, this is sound. We now define resolution — this process is central to deductive inference, and may be inverted to form a useful inductive inference procedure.

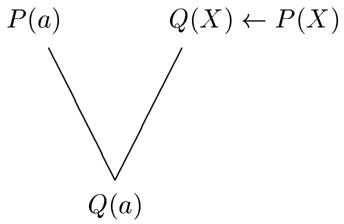
Definition 3.6.1 (resolution) *Let $C_1 = L_1 \vee L_2 \vee \cdots \vee L_i \vee \cdots \vee L_m$ and $C_2 = M_1 \vee M_2 \vee \cdots \vee M_j \vee \cdots \vee M_n$ be two clauses that are standardized apart. Let L_i and M_j be the literals resolved* *upon and let θ be a most general unifier for L_i and M_j. Then the binary resolvent is:*

$$(L_1 \vee L_2 \vee \cdots \vee L_{(i-1)} \vee L_{(i+1)} \vee \cdots \vee L_m \vee \\ M_1 \vee M_2 \vee \cdots \vee M_{(j-1)} \vee M_{(j+1)} \vee \cdots \vee M_n)\theta$$

Resolution is diagrammatically shown with a large vee.

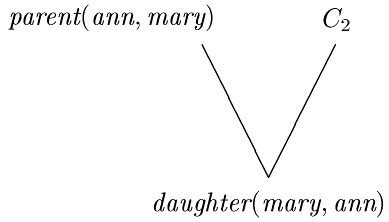


Example 3.6.1 *Consider the clauses $P(a)$ and $Q(X) \leftarrow P(X)$. The latter clause can also be written $Q(X) \vee \neg P(X)$. Now we resolve upon the literals $P(a)$ and $\neg P(X)$ where the substitution $\theta = \{X/a\}$ is a* mgu *deriving the resolvent $(Q(X))\theta = Q(a)$.*



We now discuss the inverse resolution operator referred to as the v-operator. We will later see how two v-operators can be combined to give the inverse resolution operator referred to as the w-operator. The v-operator induces C_2 from C_1 and R, such that R is the resolvent of C_1 and C_2. In general C_2 is not unique as there is a variety of choices that may be made in the algorithm. This is best illustrated with an example.

Example 3.6.2 *Suppose $C_1 = parent(ann, mary)$ and $R = daughter(mary, ann)$. Then the v-operator must induce clause C_2.*

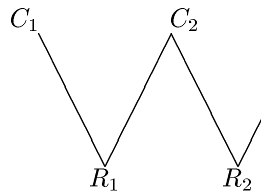


We could induce the clause $C_2 = daughter(X, Y) \leftarrow parent(Y, X)$ which would be the most useful clause to induce. Other solutions include:

$$daughter(mary, ann) \leftarrow parent(ann, mary),$$
$$daughter(X, ann) \leftarrow parent(ann, X),$$
$$daughter(mary, Y) \leftarrow parent(Y, mary)$$
$$daughter(mary, ann) \leftarrow parent(Y, mary),$$
$$daughter(mary, ann) \leftarrow parent(ann, X)$$
$$daughter(mary, ann) \leftarrow parent(Y, X).$$

In general there exists a vast range of possible answers, of which only a few will be of any value.

The w-operator consists of two binary resolution operators side by side with the middle clause in common as shown below. Given R_1 and R_2, induce C_1, C_2, and C_3, such that R_1 is the resolvent of C_1 and C_2 and R_2 is the resolvent of C_2 and C_3. The w-operator is useful as it is able to invent new predicates.



3.7 Quinlan — The FOIL System

FOIL extends techniques used in attribute value learning systems to ILP. It is a significant system because many other systems, such as MFOIL, CHILLIN, FOCL, CHAM, CHAMP, FLIP, and FOIDL developed from ideas initially presented in FOIL. Also, FOIL has consistently demonstrated its ability to efficiently induce good hypotheses. The system's simplicity and speed have made it an ideal bench mark for ILP. Also, FOIL is written in *C* which enhances both its speed and portability.

The two basic techniques used by FOIL are divide-and-conquer, such as in CLS, ID3, CHART, ASSISTANT, and the covering approach, found in the AQ family of learners.

FOIL is provided with a set of positive and negative examples of the target predicate. It is also provided with background knowledge in the form of a partial extensional description of certain background predicates. From this FOIL induces a set of clauses that is an intensional description of the target predicate. *Type* information is also provided to the learner. Note, FOIL is restricted to function free terms. *Key* information, which restricts the way literals are incorporated into a clause, is optionally provided. An outline of the algorithm for FOIL is given in Algorithm 3. The outer shell is a greedy covering algorithm (see Section 3.10).

Algorithm 4 outlines how individual clauses are induced. Each clause is initialized to:

$$P(V_1, V_2, \dots, V_n) \leftarrow$$

where P is the target predicate and the terms in the head of the clause consists of n distinct variables. This clause clearly covers all the examples including the negative ones. Literals are added to the tail of this clause until no negative examples are covered by the clause. As there is a choice of possible literals to add, a selection criterion is required. The objective of the selection criterion is to add literals that will reduce the negative coverage as much as possible while maintaining the positive coverage.

FOIL uses the information gain heuristic to compare potential literals. A list of tuples is maintained, these tuples are labelled either positive or negative. Initially, this tuple list will be identical to the list of positive and negative examples. The tuple list maintains variable bindings as new literals are added. If a literal does not cover a tuple then the tuple is removed. However, if a literal has bindings for a new variable then the tuple is replicated and given the different variable bindings, which potentially increases the size of the tuple list. The information gain heuristic measure is based on the change in the number of positive and negative tuples in this tuple list as a new literal is added to the partially completed clause. The *gain* is calculated as follows:

$$\text{gain} = p'(\lg(\frac{p_1 + 1}{n_1 + p_1 + 1}) - \lg(\frac{p_0 + 1}{n_0 + p_0 + 1})))$$

where p' is the number of positive tuples that are covered when this literal is added (this is before the tuple list is updated), p_0 and p_1 are the numbers of positive tuples before and after the new literal is added, and n_0 and n_1 are the numbers of negative tuples before and after the new literal is added. Note, if a literal just introduces a new variable and doesn't alter the tuple list then it's gain will be 0. This is undesirable as such a literal may be needed to induce the target concept. Hence, a small (relative to the gain of gainful literals) positive gain is given to these determinate literals, which biases the search to introduce new variables when no better alternatives exist.

Input :
  E^+ – Positive examples.
  E^- – Negative examples.
  B – Background knowledge.
Output :
  H – Induced Hypothesis.
begin
  $H := \{\}$
  $\mathcal{X} := E^+$
  while $\mathcal{X} \neq \{\}$ **do**
    $c :=$ induce_clause$(\mathcal{X}, E^-, B)$
    $\mathcal{X} := \mathcal{X} - \{e \in \mathcal{X} | \{c\} \cup B \models e\}$
    $H := H \cup \{c\}$
  od
  output H
end

Algorithm 3: Outline of FOIL

3.8 Muggleton — Relative Least General Generalization

GOLEM, the ILP system developed by Feng and Muggleton [90], uses Plotkin's [99] relative least general generalization technique(*rlgg*)[4]. This ILP system has been able to learn effectively in both artificial domains, including most of the simple recursive relations, and real world domains including predicting the secondary structure of proteins [54]. A brief description of *lgg* and *rlgg* will now be given before the algorithm for GOLEM is described.

[4]Plotkin introduced the *rlgg* by using a set of clauses as the training examples. These clauses consist of a head, effectively the same as a positive example, and a body which was the background knowledge pertaining to the example at the head. The *lgg* of two of these clauses has a body which is a subset of the body of the *rlgg*(used in GOLEM) of the corresponding examples.

Input :

$\mathcal{X}$ – Examples still to cover.

E^- – Negative examples.

B – Background knowledge.

Output :

b – induced clause

begin

$T := \{\langle e, \oplus\rangle | e \in \mathcal{X}\} \cup \{\langle e, \ominus\rangle | e \in E^-\}$

where $\langle e, \oplus\rangle(\langle e, \ominus\rangle)$ is a positive(negative) tuple.

$b := \{\}$

while T contains $\ominus$ elements **do**

$l :=$ find the most gainful literal with respect to T

$T :=$ exclude and expand tuples T using l

$b := b \cup l$

od

return b

end

Algorithm 4: induce_clause function

The *lgg* of two terms is defined as follows:

$$
\begin{aligned}
\operatorname{lgg}(t_1, t_1) &= t_1, \\
\operatorname{lgg}(t_1, t_2) &= V, \quad \text{where } t_1 \neq t_2 \text{ and } V \text{ is a variable} \\
&\qquad \text{which represents the lgg of } t_1 \text{ and } t_2, \\
\operatorname{lgg}(f(t_1, t_2, \ldots, t_k), f(s_1, s_2, \ldots, s_k)) &= f(\operatorname{lgg}(t_1, s_1), \operatorname{lgg}(t_1, s_1), \ldots, \operatorname{lgg}(t_1, s_1)), \text{ and} \\
\operatorname{lgg}(f(t_1, t_2, \ldots, t_k), g(s_1, s_2, \ldots, s_j)) &= V, \quad \text{where } f \neq g \vee k \neq j \text{ and } V \text{ is a} \\
&\qquad \text{variable which represents the lgg of} \\
&\qquad \text{the two function terms.}
\end{aligned}
$$

The *lgg* of two atoms is defined as follows:

$$
\begin{aligned}
\operatorname{lgg}(p(t_1, t_2, \ldots, t_k), p(s_1, s_2, \ldots, s_k)) &= p(\operatorname{lgg}(t_1, s_1), \operatorname{lgg}(t_1, s_1), \ldots, \operatorname{lgg}(t_1, s_1)), \text{and} \\
\operatorname{lgg}(p(t_1, t_2, \ldots, t_k), q(s_1, s_2, \ldots, s_j)) &= \textit{undefined}, \quad \text{where } p \neq q \vee k \neq j.
\end{aligned}
$$

The *lgg* of two literals is simply: the *lgg* of the atoms if neither literal is negated, the negation of the *lgg* of the atoms if both are negated, and undefined if one is negated and the other is not negated. We give some examples below. Let A and B be atoms. Then,

$$
\begin{aligned}
\operatorname{lgg}(\neg A, \neg B) &= \neg\operatorname{lgg}(A, B), \\
\operatorname{lgg}(\neg A, B) &= \textit{undefined}, \text{ and} \\
\operatorname{lgg}(A, \neg B) &= \textit{undefined}.
\end{aligned}
$$

The *lgg* of two clauses is defined:

$$\text{lgg}(C, D) = \{\text{lgg}(L_1, L_2) | L_1 \in C, L_2 \in D, \text{ and } \text{lgg}(L_1, L_2) \text{ is defined}\}$$

We now motivate the *rlgg* operator which is used in GOLEM. Suppose we are given background knowledge B which is finite and consists of ground atoms and we are also given two examples e_1 and e_2, then we wish to find a clause C such that both the following formulas are true: $B \wedge C \models e_1$ and $B \wedge C \models e_2$. By rearranging these formulas we find $\models C \rightarrow (e_1 \leftarrow B)$ and $\models C \rightarrow (e_2 \leftarrow B)$. These formulas will hold if C is a generalization of $e_1 \leftarrow B$ and C is a generalization of $e_2 \leftarrow B$. The *lgg* of $e_1 \leftarrow B$ and $e_2 \leftarrow B$ will generalize both of these clauses. Hence, GOLEM takes the *lgg* of these clauses which is defined to be the *rlgg* of the two examples with respect to the background knowledge. In general this *rlgg* will be large. However, GOLEM restricts this newly formed clause to relevant and determinate literals, this reduces the size of the clause significantly producing useful generalizations. A literal is considered relevant if it either contains a variable which appears in the head of the clause or it contains a variable which appears in another relevant literal. Although the *rlgg* will contain all the literals in B none of them are relevant and hence are removed. GOLEM also uses Determinacy to help tame this enormous search space(see Section 3.11.5 on Page 37).

Example 3.8.1 *Consider the family tree consisting of: Jim, Helen, Mark, and Karen. Suppose the background knowledge consists of the following ground literals: parent(Jim, Mark), parent(Helen, Mark), parent(Mark, Karen), male(Jim), male(Mark), female(Helen), and female(Karen). Given the two positive examples* $e_1 = \textit{father}(\textit{Jim}, \textit{Mark})$, $e_2 = \textit{father}(\textit{Mark}, \textit{Karen})$, *and let* B *denote the conjunction of the literals in the background knowledge.*

$$\begin{aligned}
\textit{rlgg}(e_1, e_2) = &\textit{lgg}((e_1 \leftarrow B), (e_2 \leftarrow B)) \\
= &\textit{father}(V_{JM}, V_{MK}) \leftarrow \textit{parent}(\textit{Jim}, \textit{Mark}), \textit{parent}(\textit{Helen}, \textit{Mark}), \\
&\textit{parent}(\textit{Mark}, \textit{Karen}), \textit{parent}(V_{JH}, \textit{Mark}), \textit{parent}(V_{JM}, V_{MK}), \\
&\textit{parent}(V_{HM}, V_{MK}), \textit{parent}(V_{HJ}, \textit{Mark}), \textit{parent}(V_{MJ}, V_{KM}), \\
&\textit{parent}(V_{HM}, V_{KM}), \textit{male}(\textit{Jim}), \textit{male}(\textit{Mark}), \\
&\textit{male}(V_{JM}), \textit{male}(V_{MJ}), \textit{female}(\textit{Helen}), \textit{female}(\textit{Karen}), \\
&\textit{female}(V_{HK}), \textit{female}(V_{KH}).
\end{aligned}$$

After the irrelevant and nondeterminate literals are eliminated, the following clause is generated:

$$\textit{father}(V_{JM}, V_{MK}) \leftarrow \textit{parent}(V_{JM}, V_{MK}), \textit{male}(V_{JM}).$$

Clearly, the two examples contribute to the generation of the one clause. Hence, they must be part of the same clause in the target concept. If the clauses are formed from examples that are represented by two different clauses in a target hypothesis, then, in general, the clause produced by the *rlgg* will be useless. GOLEM addresses this problem by considering a large sample of pairs to generate potential clauses using the *rlgg*, the best of these is selected and further generalized by this *rlgg* process. The best clause, the clause with no negative cover and the most positive cover, is the induced clause in the greedy cover loop. Clearly, it is infeasible to consider all pairs, hence

GOLEM randomly samples a selection of these pairs. This sampling must be sufficient such that with high probability two examples will form part of one clause in the target hypothesis.

3.9 Inverse Implication

Generalization using θ-subsumption will not always succeed even if there exists a generalization. This is because although if $C \geq D$ implies $C \models D$, the converse is not true. This means methods that rely on θ-subsumption may not always induce the desired target hypothesis [23], especially when recursive clauses are used more then once in the resolution process. By inverting implication directly the limitation inherent in θ-subsumption may be overcome.

LOPSTER [58] and CRUSTACEAN [2, 3], restrict the class of recursive logic programs they learn considering the effect of multiple resolution steps inverting implication directly. However, it is difficult to see how such an approach could be extended for a larger class of problems.

A more general approach is discussed by Muggleton [85] which considers flattened clauses and investigates the result of repeated resolution steps undertaken by the recursive clause. Although, this strategy shows promise it has yet to be implemented.

3.10 Greedy Covering Algorithm

Most ILP systems like GOLEM [90] and FOIL [104] employ a greedy covering algorithm to build hypotheses [72]. Quinlan refers to this approach as "separate and conquer" [104].

The approach works as follow:

- Find the clause that covers the maximum number of positive examples without covering any or only a few negative examples.
- This clause forms part of the induced hypothesis and the positive examples covered by it are removed, thereby creating a reduced set of positive examples.
- The above process is repeated with this new set of positive examples and the original set of negative examples until there are no uncovered positive examples.

While this approach has lead to efficient learning in many applications, there are situations in which it fails to perform well. The algorithm for this approach is given in Algorithm 5.

3.11 Dimensions of ILP systems

ILP systems may be contrasted along many different dimensions. These dimensions may be in terms of restrictions in the ILP problem, or in the techniques used by ILP systems — they are important in describing and characterizing ILP systems.

3.11.1 Top-down and bottom-up

The most common way of describing an ILP system is as either a *top-down* or a *bottom-up* system. This refers to the general direction in which the search proceeds. A top-down system will start

Input :

E^+ – Positive examples.

E^- – Negative examples.

B – Background knowledge.

Output :

H – Induced Hypothesis.

```
begin
  H := {}
  S := E+
  while S ≠ {} do
    C := induce a clause that covers as many examples as possible of S,
           and no examples from E− with respect to B
    S := S −  the positive examples covered by C w.r.t. B
    H := H ∪ C
  od
  output H
end
```

Algorithm 5: ILP Greedy covering algorithm

with a general hypothesis and use specialization to search for the target hypothesis. Whereas bottom-up systems start with a specific hypothesis and use generalization. This categorization is problematic as the system may proceed in a top-down manner in one part of the search, and, in another it may search in a bottom-up manner. For example, FOIL, when searching for individual clauses is top-down. However, it also uses the greedy covering technique, to induce the set of clauses that form the final hypothesis, which is bottom-up. In most cases this categorization refers to the search conducted on the clauses, rather than the logic programs, hence, FOIL is considered top-down.

Most ILP systems search in a top-down manner. There are a few notable exceptions such as CIGOL and GOLEM.

3.11.2 Batch learning and incremental learning

A *batch learning* system receives all the positive and negative examples at once, then it must induce a hypothesis from only this information. Whereas an *incremental learning* system receives the examples one at time and at each stage induces a hypothesis from the examples seen so far.

3.11.3 Single and multiple predicate learning

ILP systems that are able to learn more than one predicate are referred to as *multiple predicate* learners, whereas, systems that only consider one predicate are *single predicate* learners. It is possible to form a multiple predicate learner out of a single predicate learner by simply applying the single predicate learner to each predicate in turn. However, systems that handle multiple predicates internally have two main advantages. First, each predicate may use other predicates in the bodies of their clauses. An example of this is an ILP system attempts to learn both *odd* and *even* at the same time. A consistent and complete hypothesis induced by a multiple predicate learner, with *inc* as background knowledge, may be:

$$\begin{array}{l} odd(1). \\ even(X) \leftarrow inc(Y,X), odd(Y). \\ odd(X) \leftarrow inc(Y,X), even(Y). \end{array}$$

Second, multiple predicate learners may invent predicates which are common to all the predicates induced.

3.11.4 Interactive and noninteractive

The distinction between *interactive* and *noninteractive* ILP systems is as the name suggests. Interactive systems ask the teacher or operator questions during the learning process regarding the domain, whereas, noninteractive system do not require any interaction on the part of the user. For example Sammut's [115, 116] MARVIN is an interactive ILP system which asks the teacher if new examples, constructed by the program, form part of the concept. Another important interactive system is Hume's CAP which interacted with its environment by constructing experiments.

3.11.5 Language bias

As the search space is vast in first-order logic, language bias is an inescapable method used by all practical ILP systems to restrict the search space. Since this is an important dimension when describing an ILP system, language bias will be discussed in some detail. The motivation for restricting the search space is either to simplify the structure of the search space making it easier and faster to search, or to simply 'thin out' the search space making the search process faster. However, there is a cost in restricting the hypothesis search space. This cost occurs in two ways. First, the target concept may not be representable in the restricted search space, and hence, will not be induced by the learner. Second, the target concept may have a more complex representation in the restricted search space making the search more difficult and longer. The following is a list of the main language biases used in ILP:

Program clauses: A *program clause* is a clause of the form:

$$H \leftarrow L_1, \dots L_n$$

where H is an atom and L_i is of the form L or $\neg L$ where L is an atom. Program clauses are less expressive than a general formula. For example a formula such as $P(a) \vee P(b)$ may not be expressed with program clauses. Deduction based on SLD-resolution is complete for program clauses. Also, the programming language PROLOG uses program clauses and SLD-resolution for its deduction. For these reasons this bias is implicit in most ILP systems.

Determinate clauses: A program clause $A \leftarrow B_1, \dots B_n$ is *determinate* with respect to a definite logic program L if for all $1 \leq i \leq n$ and for all substitutions θ such that $(A \leftarrow B_1, \dots B_{i-1})\theta$ is ground and $L \models (B_1 \wedge \cdots \wedge B_{i-1})$, if there is a substitution δ such that B_i is ground and $L \models B_i\theta\delta$, then the substitution δ is unique.

This restriction is useful in ILP as it simplifies and speeds up the resolution process. However, is does reduce the concepts that may be represented. For example if the target concept is the *"connected"* relation of an undirected graph, where the background knowledge is *linked_to*, then such a concept is representable by a set a Horn clauses, namely:

$$\begin{aligned} &\mathit{connected}(A,B) \leftarrow \mathit{linked\_to}(A,B).\\ &\mathit{connected}(A,B) \leftarrow \mathit{linked\_to}(A,C), \mathit{connected}(C,B). \end{aligned}$$

However, it is not representable by a set of determinate Horn clauses. Although there is a reduction in concepts that may be represented, restricting to determinate clauses still allows a large class of useful concepts to be represented. The advantage gained in reducing the hypothesis space generally outweighs the cost of a reduction in concepts that may be represented, and hence, this restriction is embraced by most ILP systems.

Note, determinacy is dependent on the background knowledge and the order of the literals in the body of the clause, e.g., the clause $\mathit{add}(A,B,C) \leftarrow \mathit{inc}(D,A), \mathit{add}(D,B,E), \mathit{inc}(E,C)$ with respect to the usual logic programs for *add* and *increment* is determinate, however, the clause $\mathit{add}(A,B,C) \leftarrow \mathit{add}(D,B,E), \mathit{inc}(D,A), \mathit{inc}(E,C)$ is not.

ij-determinacy: A program clause is *ij-determinate* if it is determinate and the depth of any variable in the clause is at most i and the degree of any term is at most j. The *depth* of a variable in a determinate clause is 0 if the variable appears in the head of the clause, otherwise it is one plus the maximum depth of the variables that are used in determining its binding. The *degree* of a term is the number of other terms within the same literal that functionally determine its binding. This restriction was introduced by Muggleton and Feng [90] and used in the GOLEM ILP system. One simplification of this is to consider j to restrict the arity [61, page 112] rather than degree, although this is different, it embraces a similar class of hypotheses.

Function free: A clause is *function free* (also known as datalog) if there are no function symbols with nonzero arity. Essentially, a term is either a constant or a variable. This simplifies the structure of each clause without seriously affecting expressiveness, since, a clause which contains function symbols may always be flattened to a clause without function symbols which expresses the same relation. This requires both additional constant symbols and additional background predicates.

Constant free: A clause is *constant free* if it contains no constant symbols. This restriction is similar to function free clauses in that it simplifies the structure of each clause without affecting the expressiveness, as constants may be introduced through predicates in the background knowledge. Consider the clause :

$$\mathit{over12}(A) \leftarrow \mathit{age}(A,B), B > 12.$$

Given that the background knowledge contained the predicate *const12*(12) the *over12* clause could be expressed without constants as :

$$\mathit{over12}(A) \leftarrow \mathit{age}(A,B), \mathit{const12}(C), B > C.$$

Nonrecursive: A program clause is *nonrecursive* if its head predicate does not appear in the body of the clause. This restricts the concepts that may be represented, however it makes resolution considerably simpler and it also allows clauses to be considered more independently. Given that in many domains, especially real world domains, the concept description does not require recursion this restriction is very useful.

k-clause: A logic program is *k-clause* if it contains at most k clauses.

k-literal: A clause is *k-literal* if it contains at most k literals.

Generative: A program clause is *generative* if all the variables that occur in the head also occur in the body. These clauses are also referred to as *allowed*, *range restricted*, or *connected*. So, for example $P(A, B) \leftarrow Q_1(A), Q_2(B, C), Q_3(C)$ is generative, whereas $P(A, B) \leftarrow Q_1(A)$ is not, as, the variable B does not occur in the body of the clause.

Constrained: A program clause is *constrained* if all the variables that occur in the body also occur in the head of the clause. For example $P(A, B) \leftarrow Q_1(A)$ is constrained whereas $P(A, B, C) \leftarrow Q_1(A), Q_2(B, D), Q_3(D)$ is not. Note, neither of these are generative.

No *negation-as-failure* literals: A program clause where the body contains no *negation-as-failure* literals. This is a commonly used restriction that both simplifies the proof procedure and restricts the search space.

3.11.6 Search bias

The *search bias* relates to the way the ILP system proceeds through the search. Generally, it is not feasible to exhaustively search the entire hypothesis space, hence most ILP systems will use heuristics to direct the search and ignore "unfruitful" branches in the search tree. This affects or biases the possible logic programs induced. In general there exist hypotheses that are part of the language bias, yet are implicitly excluded from the hypothesis space by the way this space is searched.

When an ILP system is unable to induce a logic program which is representable within the *language bias* restriction, it is generally due to the search bias not considering the target hypothesis. One way of addressing this problem is to expand the search bias or language bias, known as *bias shift*.

3.11.7 Validation bias

A learner may only search the hypothesis space to a finite depth and then stop and return its induced hypothesis. The mechanism that determines this depth is know as the *stopping criterion*. This mechanism limits or biases the set of hypothesis the learner may induce, and this restriction is referred to as the *validation bias*.

3.11.8 Imperfect data

Most of the early ILP systems assumed the examples and background knowledge to be both complete and without error. This is not generally the case in many real world domains. Newer

ILP systems have addressed different aspects of this problem in a variety of ways. The way in which a system deals with imperfect data is an important dimension in describing a system.

Imperfect data may occur in the following forms:

- the examples may contain classification noise where some positive (negative) examples may be miss-labelled as negative (positive);
- individual terms in the examples may be corrupted, this is equivalent to attribute noise in propositional learning, if examples contain continuous terms then the noise may either be Gaussian or uniform;
- the examples may be sparse which cause difficulties when learning recursive relations;
- there may be missing terms;
- the background knowledge may be incomplete; or
- the background knowledge may contain irrelevant predicates.

Classification noise is the most common type of imperfect data addressed by ILP systems. If an ILP system does not handle classification noise then it will generally *over fit* the data. This produces hypotheses which contain more and longer clauses and are generally less predictive. A simplistic method of solving this problem is to tolerate a fix level of incorrectness with respect to the training data. This will work with low levels of noise, however, as the level of noise increases, more sophisticated methods that trade off accuracy and succinctness are required. One approach is to use statistical or information theoretic measures which are incorporated into the search heuristic and stopping criteria to handle the noise during the induction process. An overview of these is given by Lavrač and Džeroski [61, 62].

A compression measure may be applied to address the problem of noise. The learner induces the hypothesis that provides the greatest compression of the training data [92]. So, a noisy example will generally not be compressible and hence it is not generalized, whereas a group of 'similar' examples may be compressed into a single clause. This line of reasoning is closely related to the MML (minimum message length) [129] or MDL (minimum description length) [109] techniques. These are motivated or justified from Bayesian reasoning. An overview is given by Stahl [126].

Another approach is to preprocess the training examples, detecting noisy examples and removing them. This filters the training set, providing the learner with noise free data [29]. This may be applied to any ILP system as this filtering is independent of the learner.

Post-processing the induced hypothesis may also make an ILP system more tolerant. This approach will remove clauses that only cover a relatively small number of examples and/or remove literals that only restrict the clause from covering a few negative examples.

3.11.9 Example driven and example set driven

Another dimension or characterization of an ILP system is whether the search is driven by individual examples or by the entire example set. For example Progol uses single positive examples to seed and bias the search. This significantly reduces the search space, however, if the seed example is noisy then the subsequent search will be futile. On the other hand, Foil's search is directed by a heuristic which is calculated from all the examples. Although, the search space is not restricted by an individual example this approach is not dependent on the sampling, making it a more stable approach.

3.11.10 Predicate invention

Some ILP systems allow the automatic induction of new predicates, expanding the concepts that may be represented, this is known as *predicate invention*. Also, predicate invention may make a hypothesis more compact and hence simpler for the user to interpret it. For more information the reader is referred to Muggleton [86].

3.12 Implementations

In this section a review of a variety of ILP implementations is given. The systems are listed in approximate chronological order.

PLOTKIN'S PROGRAM: Plotkin [99] implemented an algorithm for forming generalizations of a set of datalog clauses. The program uses a heuristic approximation to form reduced lgg's and is written in the POP-2 programming language. Although, no background knowledge is used, it is partly introduced by allowing Horn clauses in the positive and negative examples. Plotkin illustrates the approach by learning the *win* position for *tic-tac-toe*. So given the positive examples:

$$\begin{array}{rcl} win(p_1) & \leftarrow & XX(1,2,p_1), XX(2,2,p_1), XX(3,2,p_1), \\ & & OO(2,1,p_1), OO(3,1,p_1), OO(1,3,p_1). \\ win(p_2) & \leftarrow & XX(1,3,p_2), XX(2,3,p_2), XX(3,3,p_2), \\ & & OO(1,1,p_2), OO(2,1,p_2), OO(1,2,p_2). \\ \vdots & & \vdots \end{array}$$

And the negative examples:

$$\begin{array}{rcl} win(p_8) & \leftarrow & XX(2,2,p_8), XX(3,3,p_8), OO(2,1,p_8). \\ win(p_9) & \leftarrow & XX(1,1,p_9), XX(1,2,p_9), XX(2,3,p_9), \\ & & OO(2,1,p_9), OO(3,1,p_9), OO(3,3,p_9). \\ \vdots & & \vdots \end{array}$$

The algorithm generalizes the positive examples inducing the solution:

$$\begin{array}{rcl} win(A) & \leftarrow & XX(B,1,A), XX(B,2,A), XX(B,3,A). \\ win(A) & \leftarrow & XX(1,B,A), XX(2,B,A), XX(3,B,A). \\ win(A) & \leftarrow & XX(1,1,A), XX(2,2,A), XX(3,3,A). \\ win(A) & \leftarrow & XX(1,3,A), XX(2,2,A), XX(3,1,A). \end{array}$$

A simplified outline of the algorithm is given in Algorithm 6.

MIS: The model inference system developed by Shapiro [122, 121] uses a refinement operator,

```
Input :
  E+ – Positive examples.
  E− – Negative examples.
Output :
  H – Induced Hypothesis.
begin
  H := E+
  while there are clauses in H that may be replaced   do
    Choose a clause D ∈ H
    Choose a clause C_i ∈ H
    Find D' = lgg(D, C_i)
    if D' is consistent with respect to E−  then
      Remove clauses in H subsumed by D'
      Add D' to H
    fi
  od
  output  H
end
```

Algorithm 6: Plotkin's Algorithm

searching top-down, to learn first-order concepts. This system and its important theoretical setting is described in detail in Section 3.5. The system was implemented in *PROLOG* with a variety of refinement operators corresponding to the class of hypotheses the system could learn. The most sophisticated could learn multiple context-free Horn sentences and term-free transformations with background knowledge. This implementation was shown to learn in a variety of artificial problems including arithmetic, list processing, unlabelled binary tree predicates, and satisfiability of boolean formulas. However, as the learning system essentially requires an oracle and does not handle noisy data, it needs some modification before it can be applied to real world domains.

MARVIN: The interactive incremental learning system MARVIN was developed by Sammut [115, 116] in the early 1980's. It introduced a simple version of inverse resolution which did not generate new variables; inverse resolution was later extended and explored by Muggleton and Buntine [88]. Inverse resolution generalizes from the training examples, effectively inverting the resolution process. MARVIN uses this generalization technique to induce a hypothesis from an initial example. This hypothesis is tested by asking the trainer if new examples, constructed by the program, form part of the concept. If the hypothesis is too general MARVIN uses specialization.

DUCE: DUCE [82] introduced by Muggleton is capable of learning propositional Horn clauses. This system makes use of six transformation operators: inter-construction, intra-construction, absorption, identification, dichotomization, and truncation. These are used to successively compress the given examples to induce the final hypothesis. DUCE's effectiveness is demonstrated in a chess end game domain. Other systems, such as REDUCE [8], have followed this approach.

CIGOL: Muggleton and Buntine [88] implemented the ILP system CIGOL which is a multiple-predicate incremental bottom-up learner. CIGOL uses inverse resolution to construct clauses from an example not implied by the current hypothesis. The system also makes use of the w-operator (see Section 3.6) to invent new predicates. CIGOL is interactive, in that it asks the user if a newly constructed clause is correct, it also asks the user to name newly invented predicates.

FOIL: Quinlan [24, 104, 25] uses techniques from his propositional learner ID3 [103] to produce the FOIL system. FOIL is a top-down, noninteractive, batch learner, which induces function-free normal programs. Using the covering approach of the AQ systems [74], FOIL induces a list of clauses one at a time attempting to cover all the positive examples and none of the negative, this is explained in more depth in Section 3.7. FOIL induces a partial ordering on the constants with respect to the background knowledge to assert the clauses it generates are recursively sound. By insisting that at least one term in a recursive predicate maintains this partial ordering over all the positive examples, no infinite loops will be induced, and hence, the clause will be recursively sound. In constructing each clause an information gain heuristic is used to direct the search. This system has become an important benchmark for ILP systems. Quinlan [104] extended FOIL to FFOIL which learns relations where the last term is a function of the previous terms. This system is also studied with respect to boosting and bagging [106].

FOCL: The FOCL learning system developed by Pazzani and Kibler [97] extends FOIL by combining an explanation base approach to learn and the first-order representation. As with FOIL, FOCL generates function-free Horn clause rules. However, FOCL also optionally allows intentionally defined predicates in the background knowledge, a set of typing constraints on the predicates, and an initial rule. These enable FOCL to focus the search, making it more efficient and effective.

GOLEM: Muggleton and Feng's [90] GOLEM uses the same greedy covering approach to FOIL, however, individual clauses are constructed bottom-up rather than top-down. GOLEM is a noninteractive, batch, single-predicate learning system. The background knowledge B is provided in the form of ground atoms. This background knowledge is used in the relative least general generalizations(RLGG) to generalize two positive examples. A RLGG of e_1 and e_2 is a least general generalization(LGG) under theta subsumption of $e_1 \leftarrow$ B and $e_2 \leftarrow$ B. To tame the size of the resulting reduced clause, this clause is restricted to the class of ij-determinate clauses, where i and j are preset parameters. This class is shown to be polynomially bound in the number of literals by i, j and $|B|$. Initially, a random selection of pairs are chosen to generalize via RLGG. These are tested for consistency against the negative examples. The best consistent clause is further generalized in a similar manner until no improvement is made. This becomes the first clause in the greedy covering approach where positive examples it covers are removed and the process is repeated. GOLEM was implemented in C and shown to work on a variety of artificial and real world domains, including the *finite-element mesh design*, *protein folding*, *quick sort*, and the *king-rook-king end game* domains.

LINUS,DINUS: The LINUS first-order learning system developed by Lavrač, Džeroski, and Gro-

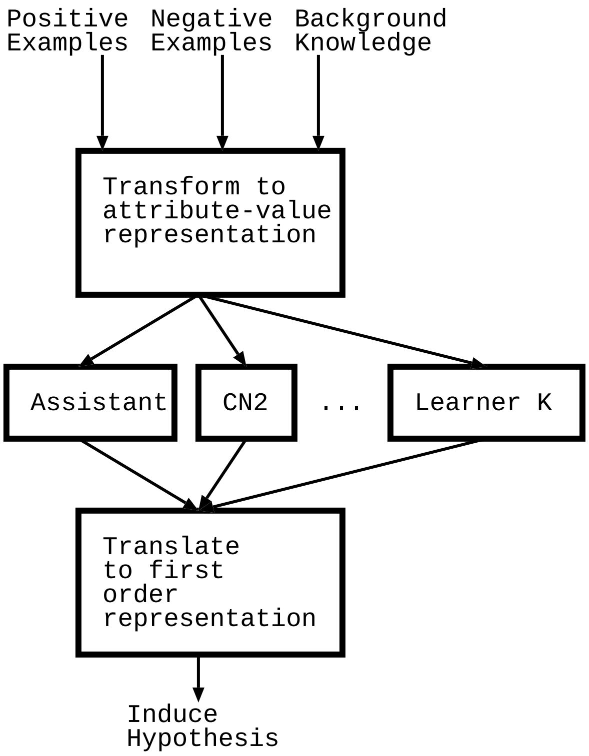


Figure 3.2: An overview of LINUS' approach

belnik [63] is a batch, single-predicate, noninteractive learner which induces nonrecursive, function free, and constrained logic programs. This approach tabulates possible constrained clauses. These are generated from the background knowledge for the positive and negative examples. This table generates the attributes which are given to one of a number of propositional learners, such as CN2 or ASSISTANT. The hypothesis generated by the propositional or attribute-value learner is then transformed back to the first-order form which is the final induced hypothesis of LINUS. Figure 3.2 gives an overview of this approach. This approach is able to handle noise and real-value attributes by the propositional learner used. DINUS is an extension of LINUS which can learn determinate logic programs. This allows the system to learn recursive and nonconstrained programs.

CLINT: The CLINT [30] system is a first order incremental learner which combines a variety of approaches. Rather than separating the background knowledge and hypothesis, this multi-predicate learner starts with a knowledge base, a definite program, which is augmented as new examples are presented to the learner. In addition to the knowledge base there is a set of clauses which are integrity constraints, where consistence within the knowledge base is asserted. An example of an integrity constraint is "a person can not be both male and female", $\neg male(X) \vee \neg female(X)$. If a negative example is implied by the knowledge base then a back tracing algorithm, similar to MIS, is used to remove the offending clause. If an example is not implied by the knowledge base then an abduction process is used which may extend the knowledge base to imply this example. If this fails then a bottom-up induction is used to extend to the knowledge base without implying any of the negative examples. This bottom-up induction is performed, within a restricted language bias L_1, by constructing a set of specific clauses with respect to the uncovered example and then generalizing by removing literals. If CLINT is unable to induce an appropriate clause within L_1 then the process is repeated in L_2, and then L_3 etc. CLINT is also able to handle multi-value logic, this addresses negation more proficiently.

CHAM,CHAMP: One problem faced by FOIL is that some literals which are critical in inducing a target clause are not gainful, and hence, may not be introduced into the clause. This problem is partly addressed in FOIL by allowing nongainful determinate literals to be included, this introduces new and possibly useful variables, however, it will not always succeed. Kijsirikul, Numao, and Shimura address this problem in CHAM [52] by labelling variables within the target as either input or output variables depending on their function within the literal. Then the search is directed with a heuristic which combines gain and likelihood, where likelihood is a measure of the distance between the variables produced by the introduced literal and the output variable of the target predicate. This helps introduce nongainful, but useful, literals by favoring literals that are "close" to the output variable of the target predicate. Note, "closeness" between two lists is measured by the length of the maximum overlapping sublist divided by the length of the maximum list. In many respects CHAM is similar to FOIL, although, it employs the refinement operators in MIS which depends on mode declarations. CHAM searches from general to specific by instantiating head variables to a functor, unifying variables, or adding a literal to a body. Kijsirikul, Numao, and Shimura [53] extend this system into CHAMP, which is able to invent new predicates without any external user aid. The predicates are invented to increase classification accuracy and reduce the size of the induced hypothesis. The process searches for a minimal set of variables that forms a relationship which discriminates between the positive and negative examples. Considering all possible subsets of the variables present is intractable the approach adopts a greedy removal algorithm. This strategy is also adopted in CHILLIN.

FLIP: Bergadano and Gunetti's [11] FLIP interactively learns functional logic programs. The algorithm employs a general extensional top-down method like FOIL, although negative examples are not explicitly provided, they are implicitly implied from the positive examples and their functional nature. Mode information is also provided and FLIP considers clauses which are consistent with respect to this mode information. By restricting the hypothesis space to functional logic programs FLIP is able to learn from fewer examples.

MISST: Kirschenbaum and Sterling's [55] MISST extends Shapiro's MIS by using a new refinement operator which uses *skeletons* to bias the search. *Skeletons* form the flow-control for the logic program, they are the basic building blocks for program development. For example, a *skeleton* to traverse a list is:

$$\begin{array}{l} traverse([]). \\ traverse([X|Y]) \leftarrow traverse(Y). \end{array}$$

To tame the number of potential *skeletons* MISST uses meta-knowledge consisting of mode information, type information, and information about incompatibility of predicate combinations.

mFoil: The first-order system mFoil [34, 35] extends Foil by incorporating more robust noise-handling mechanisms. The search space of mFoil is extended by allowing nonground and nonunit clauses in the background knowledge. Training examples are still restricted to ground facts. The type and mode information regarding the background knowledge is similarly explicitly provided. The entropy-based information gain heuristic in Foil is replaced by either the Laplace error estimate or the m-estimate. Also, the stopping criterion uses a significance test similar to CN2 [26] and removes clauses that are not accurate enough.

Mobal: Kietz, Klingspor, Morik, Sommer, Wrobel [79, 80] developed the multi-strategy learning system Mobal which integrates various tools using a uniform representation. One of the tools is the rule discovery tool (RDT) [50] which is a batch multiple-predicate first order learner, which induces function free normal programs using extensional background knowledge. RDT uses a top-down strategy similar to Foil, however, the search space is restricted by rule schemas, predicate topology, and type information. The rule schemas describe the form a clause may take. They are simply clauses with variable predicate names. Consider the schema $\Phi(A, B) \leftarrow \Psi(A, C), \Psi(C, B)$ where Φ is replaced by "*plus2*" and Ψ is replaced by "*inc*" this will generate the clause $plus2(A, B) \leftarrow inc(A, C), inc(C, B)$. A predicate topology is also used to restrict these predicate name substitutions such that the predicate names of the body substitutions must be sub-node or equal to the predicate name of the head literal. Type information is also used to bias the search asserting all the clauses are consistent with respect to the type information. This type information is automatically learnt by the sort taxonomy tool.

Lopster: The Lopster relational learning system [58] learns a class of recursive logic programs very efficiently with a small number of examples. The class of logic programs is either restricted to purely recursive logic programs whose recursive clause has the form:

$$p(t_1, \ldots, t_n) \leftarrow p(V_1, \ldots, V_n)$$

or restricted to left-recursive logic programs consisting of a single recursive clause of the form:

$$p(t_1, \ldots, t_n, A_1, \ldots, A_m) \leftarrow p(V_1, \ldots, V_n, A_1, \ldots, A_m), L_1, \ldots, L_k.$$

where A_i, B_i, and V_i are variable and for $1 \leq i \leq m$ the variable V_i occurs in t_i, and L_i are literals that appear in the background knowledge. This includes recursive logic programs such as: *member*, *append*, *multiply*, *add*, *factorial*, and *reverse*. Although, more complex programs such as *quicksort* are beyond Lopster's capability. This system addresses a short coming of θ-subsumption which only allows the clause to be used once in a resolution step. However, Lopster requires the base case to be explicitly provided, hence, it performs poorly when given an arbitrary set of examples.

Claudien: De Raedt and Bruynooghe [31] developed the clausal discovery system Claudien which is a top-down noninteractive batch learner. Using only positive examples the system uses a downward refinement operator to find the most general clause true for each of the positive examples. Unlike most ILP systems CLAUDIEN induces full, rather than definite or normal program clauses.

CRUSTACEAN: This system is developed by Aha, Lapointe, Ling, and Matwin [2, 3], to induce small recursive programs from a small number of examples. CRUSTACEAN inverts implication, learning a class of purely recursive logic programs with single base cases and single pure recursive clauses of the form:

$$p(t_1, t_2, \dots, t_n).$$
$$p(s_1, s_2, \dots, s_n) \leftarrow p(t'_1, t'_2, \dots, t'_n).$$

where p is a predicate symbol, t_i, s_i, and t'_i are arbitrarily complex terms and t'_i is a subterm of s_i for each $i \in \{1 \cdots n\}$. LOPSTER requires the base case and performs poorly when arbitrary examples are selected, this problem is addressed in CRUSTACEAN by constructing pairs of legal base cases and generative terms which are checked against the examples. CRUSTACEAN is shown to be complete with respect to the the above class logic programs, although it is less efficient than LOPSTER.

TRACY: Bergadano and Gunetti [12] developed the ILP system TRACY which works by tracing derivations of positive examples. Unlike most ILP systems, TRACY does not induce the hypothesis one clause at a time. Rather, TRACY considers the entire set of clauses using a *PROLOG* interpreter generating consistent and complete hypothesis. *A prior* information is given to the learner that forms a tight restriction on the hypothesis space, in fact, the more efficient version of TRACY uses a finite hypothesis space which is generated at the start of the algorithm. TRACY induces its hypothesis by generating a candidate set of clauses then with each of the positive examples tracing which clause's derive the positive example. Negative examples in conjunction with a backtracing algorithm are used to remove clauses that are too general. If this fails to produce a consistent and complete hypothesis across all the positive examples the set of candidate clauses is extended and the process is repeated. TRACY allows intentional background knowledge and can induce the target hypothesis from a small number of examples. However, the *a prior* restriction on the hypothesis spaces, required to tame the search, limits the concepts that may be induced.

CHILLIN: Zelle, Mooney, and Konvisser [134] by combining top-down(similar to FOIL) and bottom-up(similar to GOLEM) techniques developed an effective ILP system, CHILLIN. At the core of CHILLIN is a compaction algorithm which starts with the most specific hypothesis, that is the set of positive examples, and then it iteratively generalizes parts of this set compacting the entire set. This compaction is directed by what compacts the most and is complete when no further compaction is possible. The final compacted set forms the induced hypothesis. In each compaction step pairs of clauses from the current set of clauses are selected and generalized. The best generalization is chosen for this compaction step. Generalizations of a pair of clauses are formed by taking the LGG of them. Note, this is considerably more efficient than GOLEM's RLGG as it does not involve the background knowledge. If this clause is consistent with respect to the negative examples, then it forms the generalization.

If it is not consistent then the clause is made more specific, in a similar manner to FOIL, by adding literals to the end of the clause. CHILLIN ensured that the recursive clauses that it generates are well founded by only adding recursive literals to the body of a clause where at least one term is a subterm of the corresponding term in the head of the clause. Note also that recursive clauses are tested within the current set of compacted clauses, so once the base cases are induced, the recursive clause will more accurately reflect its coverage. As well, CHILLIN is able to invent new predicates which is carried out in a manner analogous to CHAMP.

SPECTRE: This batch single-predicate noninteractive top-down learner developed by Boström and Idextam-Almquist [20, 19] uses unfolding and clause deletion to induce a definite program.

FOIDL: Quinlan's FOIL is extend by Mooney and Califf in FOIDL [77, 78]. This system allows intensional background knowledge and it does not require explicit negative examples. Also, FOIDL learns a first-order decision list, this is an ordered set of clauses each ending with a *PROLOG* cut. Mooney and Califf demonstrate the system by learning the past tense of English verbs.

PROGOL: The PROGOL system developed by Muggleton [94] is a multi-predicate, batch, noninteractive first-order learning system that uses the greedy covering approach in inducing the hypothesis. With the covering outer loop PROGOL uses a mix of top-down and bottom-up searches to induce the single clause for each iteration. Using inverse entailment this approach generates bottom-clauses from the background knowledge, user defined mode information, and seed examples. These are then searched from general to specific, like MIS or FOIL. Although, the space is restricted by the bottom clause, through the sub-lattice of single clause hypothesis. PROGOL uses its own sound refinement operator, searching the the subsumption lattice with an $A^\star$-like algorithm finding the clause with maximal Occam compression. PROGOL is an effective learner in a variety of domains, including the carcinogenesis domain [123].

Chapter 4

Noise model and the Bayesian Heuristic

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

4.1 Framework

Let X denote a countable class of instances. Let D_X be a probability measure on X. Let $\mathcal{C} \subseteq 2^X$ be a countable concept class. Let $D_\mathcal{C}$ represent a probability measure on 2^X. Let H be a hypothesis space and P be a probability measure (prior) over H. The concept represented by a hypothesis $h \in H$ is referred to as the extension of h (written: $ext(h)$). Further, let $\mathcal{C}$ and H be such that:

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

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

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 p positive examples and n negative examples (the reader should note that in the fixed example model, p and n are independent of the concept C).

Each of the p positive examples are generated as follows: With probability ϵ, an instance is randomly chosen 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 4.1 illustrates this process by showing how a single positive example is generated. The diagram provides the different stages in this process. Initially a Bernoulli trial is used to determine if either the example is not noisy(the left side of the diagram is taken) or possibly noisy(the right side is taken). If the example will be not be noisy then D_X is used to generate instance e. This is repeated(hence the loop in the diagram) until the instance is a noise free positive example, that is $e \in C$. If the right side is taken and the example will be possibly noisy then the positive example is generated using D_X.

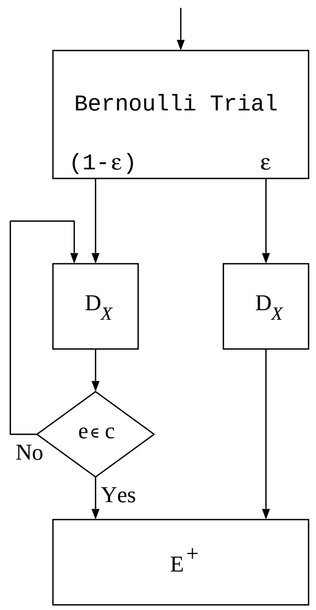


Figure 4.1: Model of positive example generation

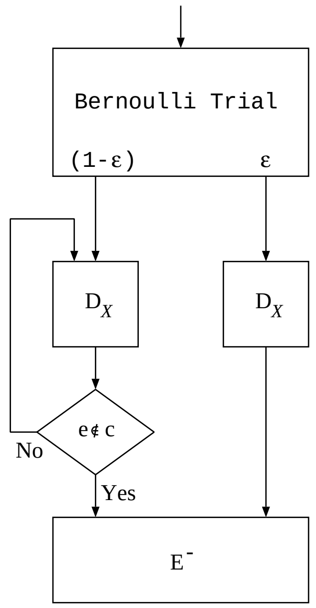


Figure 4.2: Model of negative example generation

The generation of negative examples is done similarly[1]. Figure 4.2 illustrates this process.

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.

4.2 Q heuristic

We now derive a Bayesian heuristic for finding the most probable hypothesis h given the example set E.[2] To make the algorithm deterministic we select from the most probable hypotheses the hypothesis with the minimum index. This induction can be formally expressed as follows.[3]

$$h_{\text{induced}} = \text{minindex}(\text{argmax}_{h \in H} P(h|E)) \tag{4.1}$$

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

$$P(h|E) = \frac{P(h)P(E|h)}{P(E)} \tag{4.2}$$

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.

$$P(e|h) = \begin{cases} \frac{D_X(e)(1-\epsilon)}{\theta(ext(h))} + D_X(e)\epsilon, & \text{if } e \in ext(h); \\ D_X(e)\epsilon, & \text{if } e \notin ext(h). \end{cases} \tag{4.3}$$

A few words about the above equation are in order. Given that h represents the target concept, the only way in which $e \notin ext(h)$ is if the right hand path in Figure 4.1 was chosen. Hence, in this case the conditional probability of e given h is $D_X(e)\epsilon$. On the other hand, if $e \in ext(h)$ then either the left or right hand paths in Figure 4.1 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

[1] 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.

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

[3] The notation $\text{argmax}_{h \in H} P(h|E)$ denotes the set $\{h \in H | (\forall h' \in H)[P(h|E) \geq P(h'|E)]\}$ and the notation minindex(S) denotes the hypothesis $h \in S$ such that $(\forall h' \in S - \{h\})[h < h']$, where $<$ is a total ordering on H.

is guaranteed to be from the target concept; hence $D_X(e)(1-\epsilon)$ is divided by $\theta(ext(h))$—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.

$$P(e|h) = \begin{cases} \frac{D_X(e)(1-\epsilon)}{1-\theta(ext(h))} + D_X(e)\epsilon, & \text{if } e \notin ext(h); \\ D_X(e)\epsilon, & \text{if } e \in ext(h). \end{cases} \tag{4.4}$$

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

$$P(E|h) = \prod_{e \in E^+} P(e|h) \prod_{e \in E^-} P(e|h) \tag{4.5}$$

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

Substituting 4.3 and 4.4 into 4.5 and using TP, TN, and FPN, we get the following.

$$\begin{aligned} P(E|h) = & \left(\prod_{e \in E^+ \cup E^-} D_X(e)\right) \left(\frac{1-\epsilon}{\theta(ext(h))} + \epsilon\right)^{|\mathrm{TP}|} \\ & \left(\frac{1-\epsilon}{1-\theta(ext(h))} + \epsilon\right)^{|\mathrm{TN}|} \epsilon^{|\mathrm{FPN}|} \end{aligned} \tag{4.6}$$

Now substituting 4.6 into 4.2 and 4.2 into 4.1 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.

$$\begin{aligned} Q(h) \quad = \quad & \lg(P(h)) + |\mathrm{TP}| \lg\left(\frac{1-\epsilon}{\theta(ext(h))} + \epsilon\right) + \\ & |\mathrm{TN}| \lg\left(\frac{1-\epsilon}{1-\theta(ext(h))} + \epsilon\right) + |\mathrm{FPN}| \lg(\epsilon) \end{aligned} \tag{4.7}$$

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 5.7 and 5.8, respectively.

Note, if the noise level is set to zero, $\epsilon = 0$, and no negative examples are given to the learner then the heuristic simplifies to $Q(h) = \lg(P(h)) + p\lg(\frac{1}{\theta(ext(h))})$. This is identical to the heuristic reported and incorporated into a version of PROGOL by Muggleton [87].

4.3 Theory

A number of theoretical results can be shown about this heuristic. The following proposition formalizes the intuitive expectation that if the proportion of a concept to the instance space is small, then positive examples are more useful than negative ones.

Suppose the concept only covers a small proportion of the instance space distribution. Then positive examples are of more 'value' than negative ones in the sense that they yield higher confidence in the target hypothesis than negative examples would. The counterpart to this is also true — negative examples are of more 'value' when the concept covers a large proportion of the instance space distribution.

Proposition 4.3.1 *Suppose c is a target concept and h_c is a hypothesis that represents c. Let $E = (E^+, E^-)$ be the given sets of positive and negative examples. Now, from E construct two new example sets $E_1 = (E^+ \cup \{e_1\}, E^-)$ and $E_2 = (E^+, E^- \cup \{e_2\})$ where e_1 is an instance from the concept c and e_2 is not. If $\theta(c) < 0.5$ then $P(h_c|E_1) > P(h_c|E_2)$.*

PROOF:

Let H_1 be the entropy of the positive and negative examples. Consider the ratio:

$$\begin{aligned}
\frac{P(h_c|E_1)}{P(h_c|E_2)} &= \frac{P(E_1|h_c)P(h_c)P(E_2)}{P(E_1)P(E_2|h)P(h_c)} \\
&= \frac{P(E_1|h_c)P(E_2)}{P(E_1)P(E_2|h)} \\
&= \frac{P(e_1|h_c)P(E_2)}{P(E_1)P(e_2|h)} \\
&= \frac{P(e_1|h_c)2^{-H_1(|E^+|+|E^-|+1)}}{P(e_2|h)2^{-H_1(|E^+|+1+|E^-|)}} \\
&= \frac{P(e_1|h_c)}{P(e_2|h_c)} \\
&= \frac{D_X(e_1)(\frac{1-\epsilon}{\theta(h_c)}+\epsilon)}{D_X(e_2)(\frac{1-\epsilon}{1-\theta(h_c)}+\epsilon)} \\
&= \frac{(\frac{1-\epsilon}{\theta(h_c)}+\epsilon)}{(\frac{1-\epsilon}{1-\theta(h_c)}+\epsilon)}
\end{aligned}$$

Note, a good approximation of $P(E)$, where $|E^+|+|E^-|$ is large, is $2^{-H_1(|E^+|+|E^-|)}$ [120]. Now if $\theta(h_c) < 0.5$ then $\frac{(\frac{1-\epsilon}{\theta(h_c)}+\epsilon)}{(\frac{1-\epsilon}{1-\theta(h_c)}+\epsilon)} > 1$ and hence $P(h_c|E_1) > P(h_c|E_2)$. ■

A similar result can be established about the importance of negative examples over positive examples for concepts, c, with $\theta(c) > 0.5$.

The next proposition gives a bound on the search space for the most probable hypothesis. This is very useful as a learning system can know when to stop searching, and that it has found the hypothesis h that maximizes Q across the entire hypothesis space H, even when H is infinite.

Proposition 4.3.2 *Suppose $h_t = \text{minindex}(\text{argmax}_{h \in H} Q(h))$. Then for any $h' \in H$,*

$$2^{Q(h')-|E^+|\lg(\frac{1-\epsilon}{\psi(E^+)}+\epsilon)-|E^-|\lg(\frac{1-\epsilon}{\psi(E^-)}+\epsilon)} \leq P(h_t)$$

where $\psi(E) = \min(\{D_X(e)|e \in E\} \cup \{\frac{1}{2}\})$ is the smallest probability according to the instance space distribution of examples from E. (If this value is greater the $\frac{1}{2}$ then $\psi(E)$ is set to $\frac{1}{2}$.)

PROOF: Suppose D_X is the instance space distribution and we are given the positive examples E^+ and the negative examples E^-. For any $h' \in H$ we may evaluate $Q(h')$. As $h_t = \text{minindex}(\text{argmax}_{h \in H} Q(h))$, $Q(h_t) \geq Q(h')$ and hence

$$\lg(P(h_t)) \geq \begin{array}{l} Q(h') - |\text{TP}_{h_t}|\lg(\frac{1-\epsilon}{\theta(ext(h_t))}+\epsilon) \\ -|\text{TN}_{h_t}|\lg(\frac{1-\epsilon}{1-\theta(ext(h_t))}+\epsilon) - |\text{FPN}_{h_t}|\lg(\epsilon) \end{array}$$

As $\theta(ext(h_t))$ is unknown the proof is divided into three cases.

Case 1. $\theta(ext(h_t)) < \psi(E^+)$

If $\theta(ext(h_t)) < \psi(E^+)$ then $|\mathrm{TP}_{h_t}| = 0$ and hence

$$\begin{aligned}
\lg(P(h_t)) &\geq Q(h') - |\mathrm{TN}_{h_t}| \lg(\tfrac{1-\epsilon}{1-\theta(ext(h_t))} + \epsilon) - |\mathrm{FPN}_{h_t}| \lg(\epsilon) \\
&\geq Q(h') - |E^-| \lg(\tfrac{1-\epsilon}{1-\theta(ext(h_t))} + \epsilon) \\
&\geq Q(h') - |E^-| \lg(\tfrac{1-\epsilon}{\psi(E^-)} + \epsilon) - |E^+| \lg(\tfrac{1-\epsilon}{\psi(E^+)} + \epsilon) \\
&\quad \text{as } \psi(E^-) \leq 1 - \theta(ext(h_t)).
\end{aligned}$$

Case 2. $\psi(E^+) \leq \theta(ext(h_t)) \leq 1 - \psi(E^-)$

$$\begin{aligned}
\lg(P(h_t)) &\geq Q(h') - |E^+| \lg(\tfrac{1-\epsilon}{\theta(ext(h_t))} + \epsilon) - |E^-| \lg(\tfrac{1-\epsilon}{1-\theta(ext(h_t))} + \epsilon) \\
&\geq Q(h') - |E^+| \lg(\tfrac{1-\epsilon}{\psi(E^+)} + \epsilon) - |E^-| \lg(\tfrac{1-\epsilon}{\psi(E^-)} + \epsilon) \\
&\quad \text{as } \psi(E^+) \leq \theta(ext(h_t)) \text{ and } \psi(E^-) \leq 1 - \theta(ext(h_t)).
\end{aligned}$$

Case 3. $\theta(ext(h_t)) > 1 - \psi(E^-)$

If $\theta(ext(h_t)) > 1 - \psi(E^-)$ then $|\mathrm{TN}_{h_t}| = 0$ and hence

$$\begin{aligned}
\lg(P(h_t)) &\geq Q(h') - |\mathrm{TP}_{h_t}| \lg(\tfrac{1-\epsilon}{\theta(ext(h_t))} + \epsilon) - |\mathrm{FPN}_{h_t}| \lg(\epsilon) \\
&\geq Q(h') - |E^+| \lg(\tfrac{1-\epsilon}{\theta(ext(h_t))} + \epsilon) \\
&\geq Q(h') - |E^+| \lg(\tfrac{1-\epsilon}{\psi(E^+)} + \epsilon) - |E^-| \lg(\tfrac{1-\epsilon}{\psi(E^-)} + \epsilon) \\
&\quad \text{as } \psi(E^+) \leq \theta(ext(h_t)).
\end{aligned}$$

In each case $\lg(P(h_t)) \geq Q(h') - |E^+| \lg(\frac{1-\epsilon}{\psi(E^+)} + \epsilon) - |E^-| \lg(\frac{1-\epsilon}{\psi(E^-)} + \epsilon)$ and hence

$$P(h_t) \geq 2^{Q(h') - |E^+| \lg(\frac{1-\epsilon}{\psi(E^+)} + \epsilon) - |E^-| \lg(\frac{1-\epsilon}{\psi(E^-)} + \epsilon)}$$

■

Once the value of $Q(h)$ is known for any $h \in H$, the system has a bound on the prior probability of the most probable hypothesis, thereby restricting search to a finite well defined set of hypotheses. Since there are at least three such readily available hypotheses — the hypothesis that entails the entire instance space, the hypothesis that entails no member of the instance space, and the hypothesis that exactly entails the positive examples — such a bound can be determined.

4.4 Discussion

Gold [42] showed that it is impossible to learn in the limit the class of super-affine concepts from only positive examples. Yet, there are many real situations where learning takes place from only positive examples. An example of this natural language acquisition. Children are presented with mostly positive examples of the language, and only a tiny proportion of all possible utterances form an acceptable spoken discourse. This suggests that the requirement in the Gold's model that the learner is successful on each order of presentation of the data is a very strict requirement. In reality the learner need not succeed in rare and complex data presentations. For this reason, a better setting for induction would be for the data to be generated by a stochastic process under some instance distribution.

This approach also facilitates modelling of noise in data in a very general fashion for a number of real-world scenarios. The design of learning systems that handle noise effectively, however, requires the learner to consider complete hypotheses, and hence the standard practice in ILP of constructing a hypothesis clause by clause using the covering approach is not appropriate. For this reason LIME considers entire logic programs as hypotheses.

Recently, as already noted in the a previous section, we have become aware of the work of Muggleton [87] in which a Bayesian heuristic has been used for learning from only positive data. Interestingly, if we take the noise level to be 0 in our model and only consider positive examples, then our heuristic becomes identical to Muggleton's.

Finally, we would like to note that the treatment of noise in our framework has some similarities to that of Angluin and Laird [6]. 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 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 thesis always used a noise parameter of 10% in computing $Q(h)$ even if the actual noise in the data was considerably higher.

Chapter 5

LIME

LIME is an ILP system that induces function free determinate logic programs as hypotheses from a set of ground examples and extensional background knowledge. It incorporates techniques such as the Q heuristic, described in the previous chapter, to handle noise and fixed example size, and preprocessing of the input examples to acquire information to restrict the search space. This chapter describes the LIME system by specifying the algorithm and discussing the motivation behind its design strategy.

A search engine is at the core of any ILP implementation. A naive search that does not embrace the structure of the search space will not be efficient enough to be useful. However, by utilising the structure in the search space it is possible to build a search engine that is efficient enough to be practical. This is demonstrated by many ILP systems such as FOIL, GOLEM, and PROGOL.

Clausal logic, the search space we are concerned with, has a complex but clearly defined structure. There is also considerable redundancy within clausal logic. LIME takes advantage of this structure by removing many of the redundancies, thereby constraining the search space. LIME partitions the search process into four stages: structural decomposition, 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 on to the next stage. An diagrammatic outline of LIME's approach is seen in Figure 5.1.

5.1 Hypothesis Space

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*[1]: although this restricts expressiveness, clause handling is simplified as each variable may only be bound in a unique way, and

[1] 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 Section 3.11.5 for a more formal definition.

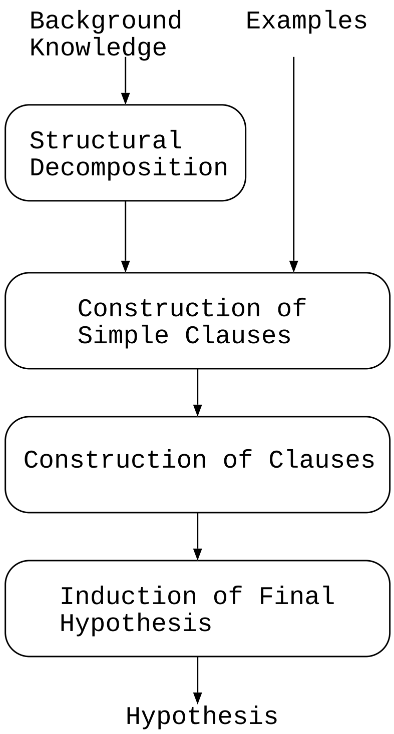


Figure 5.1: The partitioning of LIME's stages.

- 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.

5.2 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. Such lists give rise to clauses referred to 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[2] for *teen_age_boy* is:

$$teen\_age\_boy(A) \leftarrow male(A), age(A, B), B > 12, 20 > B.$$

[2] 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.

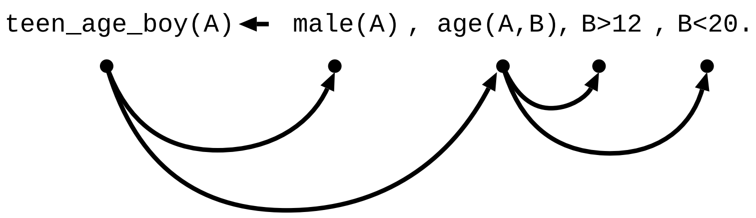


Figure 5.2: Directed graph corresponding to the example clause

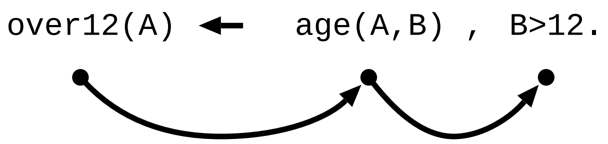


Figure 5.3: Directed graph corresponding to the over twelve clauses

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

$$\begin{aligned} &teen\_age\_boy(A) \leftarrow male(A). \\ &teen\_age\_boy(A) \leftarrow age(A,B), B > 12. \\ &teen\_age\_boy(A) \leftarrow age(A,B), 20 > B. \end{aligned}$$

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.

Definition 5.2.1 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 shown in Figure 5.2. Since it has three sink nodes, it is not a simple clause. However, the directed graph for the clause describing `over12`, shown in Figure 5.3, has exactly one sink node and hence is a simple clause.

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 $c_2 = h_2 \leftarrow b_2$. 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 $\mathcal{S} \subseteq \mathcal{C}$.

Proposition 5.2.1 *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 $\{h \leftarrow b_1, h \leftarrow b_2, \ldots, h \leftarrow b_n\}$ 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 5.2.2 *Let $c \in \mathcal{C}$ be given. Let the set of simple clauses $\{s_1, s_2, \ldots, s_n\}$ 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 $\{s_1, s_2, \ldots, s_n\}$.*

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 5.2.3 *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.

We now show that Algorithm 7 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_clauses*'. 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

Input :

BGinfo – *type* and *mode* information.

Output :

$\{C_1, C_2, \dots\}$ – A list of simple clauses.

```
begin
  current_simple_clauses := {h ← .}
  output  h ← .
  loop do
    N := {}
    foreach S ⊂ current_simple_clauses  do
      foreach l ∈ 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_clauses := current_simple_clauses ∪ N
  od
end
```

Algorithm 7: Simple Clause Enumeration.

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 nonsimple clauses may be avoided. These techniques are employed by LIME.

5.3 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.

In the following subsections we briefly discuss these aspects of the preprocessing phase as it relates to LIME. Appendix C address automatic extraction of information from the background knowledge that may be use in a variety of ILP systems.

5.3.1 Extracting type

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 [68].

Consider the background knowledge which includes the predicates *age* and *greater* (e.g. $age(jim, 27)$, $greater(5, 2)$). Clearly *age* has the type *name* and *integer* for the first and second terms respectively. Whereas *greater* has type *integer*, *integer*. Now consider learning the predicate *older* which has type *name*, *name*. A clause such as :

$$older(A, B) \leftarrow age(A, B).$$

does not need to be considered in the search as variable B is inconsistent with the type information. That is, variable B has type *name* in the head, and B has type *integer* in the *age* literal.

5.3.2 Extracting mode information

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 [68]. Another system that addresses these issues is MOBAL [81].

5.3.3 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.

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

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.

5.3.4 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.

For example three hash tables are used for storing the *inc* predicate. One hash table is required for each of the following types of queries: only the first term is bound(e.g. $inc(4, X)?$), only the second term is bound(e.g. $inc(X, 7)?$), and both terms are bound(e.g. $inc(2, 5)?$). If the query $inc(2, 5)?$ is issued then the terms 2 and 5 are combined using a hashing function and the third of the hash tables for *inc* is indexed to determine if $inc(2, 5)$ is part of the background knowledge. Note that, this does not require a linear search through the background knowledge.

5.4 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.[3] 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 5.4. This table captures the snapshot when LIME is in the process of learning the *add* relation. The two tables, *Simple Clause*

[3]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).

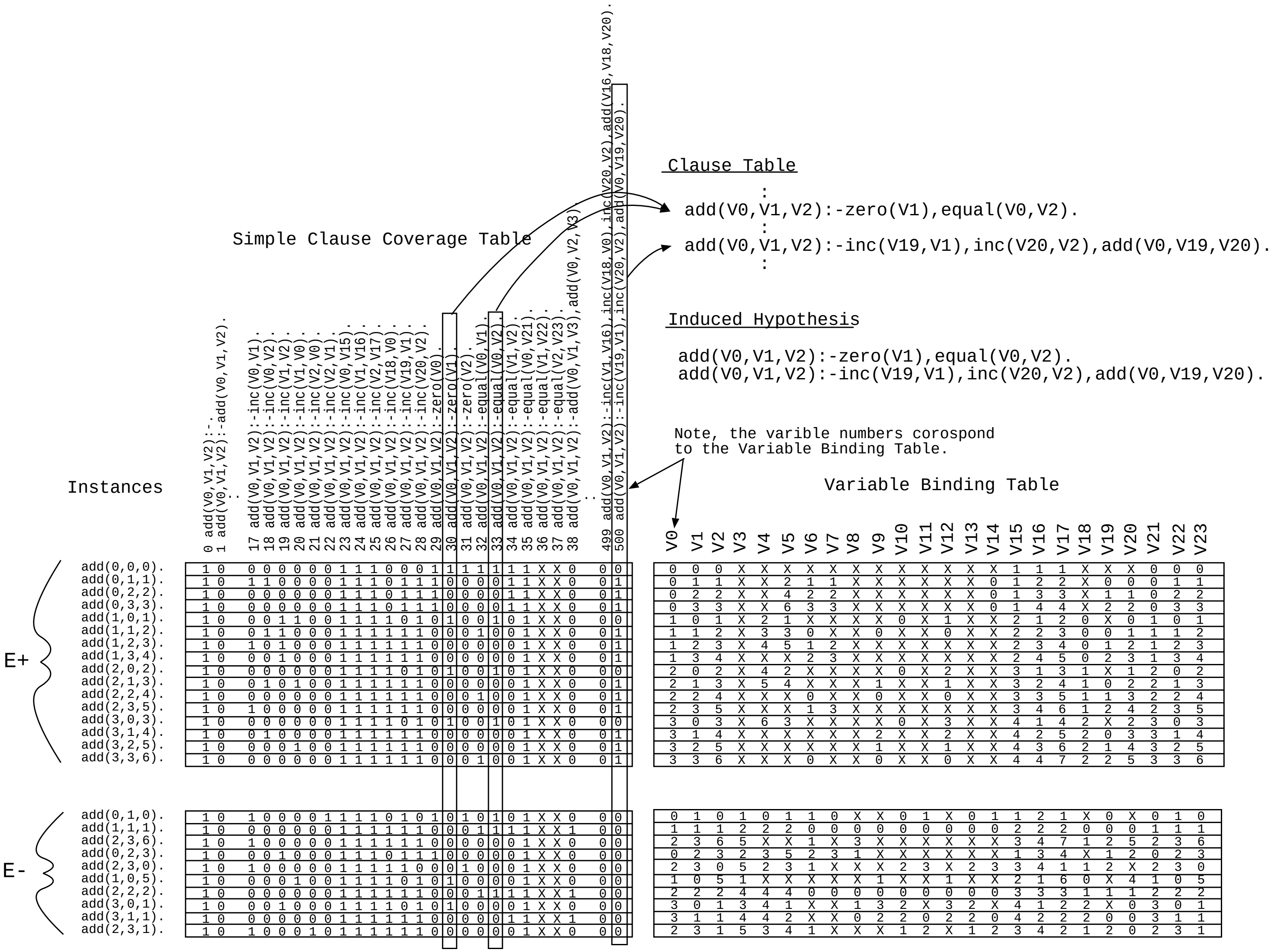


Figure 5.4: Tables constructed in the simple clause stage for *add*

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:

$$add(V0, V1, V2) : -zero(V0), equal(V0, V2).$$

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* 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:

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

By constructing the simple clauses first and then forming c_1 by combining them, c_1 is 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.

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

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[4] Any clause that contained these literal would 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.

5.4.1 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:

5.5 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.

[4]The clause should also cover no examples used in the theta estimation (see Section 5.8). Otherwise, this constraint would fail when the system is learning from only negative 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.

```
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 ← {} /* h is the head of the target predicate definition */
  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
          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
```

Algorithm 8: Simple Clause Table Construction.

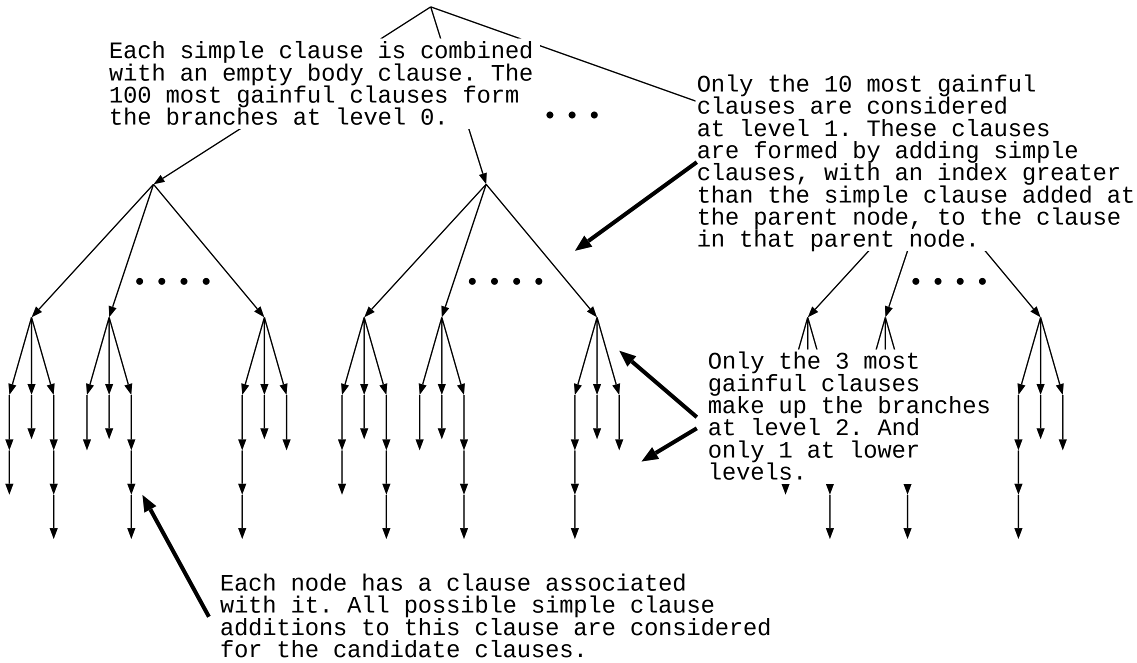


Figure 5.5: The search tree where gain_list_length = 100

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.[5] 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 5.5 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[6]. 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.

[5]Earlier implementations have looked at best-first [21, 108] and depth-bounded discrepancy search [132], but depth-first was found to be the most effective.

[6]This is a decimal number

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. Obviously, a clause that covers no examples, when combined with another simple clause, will result in a clause that also does not cover any examples and hence is of little value. 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.

Note, the algorithms objective is to obtain the best '*num_candidate_clauses*' clauses, so a list is maintained of the best clauses so far. The length of this list is bound to '*num_candidate_clauses*'. If the list has reached its bound then it is only necessary to add clauses that are better than the worst clause in the list. Thus, if a node in the search tree is reached such that any clause with more simple clauses combined with it will at best be worse than the worst clauses in the candidate clauses list then it is unnecessary to extend this branch in the search tree.

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 Chapter 4. This best estimate is achieved by assuming that other clauses that make up the final hypothesis, which is a set of clauses, are as good as this clause in terms of positive coverage, accuracy, and prior probability.

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 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.

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 $c \in S$ **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

Algorithm 9: Clause Construction from Simple Clauses.

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
             the worst clause in C) ∨ (C list not full) ]  then
         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
```

Procedure 1: Probe simple clause

calc_gain_list_size($level$,$length$)

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

Procedure 2: Calculate gain list size.

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) × (lg(p_new + 2))
```

Procedure 3: Evaluate gain of a clause.

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 $TP \times (\lg(\frac{1-noise}{theta} + noise)) +$
$(NN - FN \times S) \times (\lg(\frac{1-noise}{1-theta} + noise)) +$
$(FN \times S) \times (\lg(noise)) +$
$\text{prob}(clause) \times S$

Procedure 4: Estimate Q value of a single clause.

5.5.1 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, which, is described in many elementary texts on artificial intelligence, such as Bratko's *Prolog Programming for Artificial Intelligence* [21, page 278] or Rich's *Artificial Intelligence* [108, page 78]. 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 [26] in CN2 is not as effective as depth search.

The other approach attempted was a depth-bounded discrepancy search [132]. 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.

5.5.2 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.

5.6 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 SLD-resolution engine is used to accurately evaluate the hypothesis cover. Fourth, as the interpreter used has no backtracking, order is important in the list of clauses induced. The details are given in algorithm 10.

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

Algorithm 10: Induction of Final Hypotheses from Candidate Clauses.

probe_clause($\boldsymbol{C}, \boldsymbol{H}, \boldsymbol{R}, \boldsymbol{L}$)

Let GL be a new list of size **calc_gain_list_size**(L, *gain_list_length*)
/* **calc_gain_list_size** is defined in Algorithm 9 */
foreach $i \in \{0 \ldots |C| - 1\}$ **do**
Let NL be a record representing a new logic program
Initialize NL as follows
NL.clauses := $\{C[i].clause\} \cup R.clauses$
NL.bitvector := $C[i].bitvector \vee R.bitvector$
if (NL covers some elements) $\wedge$ (NL.bitvector $\neq$ R.bitvector) **then**
gain := **gain_lp**(R, NL)
if [(extensions to NL could prove better than the worst clause in H) $\vee$ (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 the SLD-resolution engine and NL.*clauses*
/* If NL.clauses is recursive then NL.bitvector (and *value*) */
/* are only estimates; therefore, the SLD-resolution engine is used */
/* to compute the actual coverage and *value*. */
value := **bay**(NL)
Add (NC, *value*) to the list H
fi
fi
od
foreach $GE \in GL$ **do**
probe_clause($C, H, GE, L + 1$)
od

Procedure 5: Probe clause.

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 $(\lg(p\_new + 2)) \times (2 \times (n\_old - n\_new) + (p\_new - p\_old))$

Procedure 6: Gain measure for a logic program.

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 $TP \times (\lg(\frac{1-noise}{theta} + noise))+$
$TN \times (\lg(\frac{1-noise}{1-theta} + noise))+$
$(FP + FN) \times (\lg(noise))+$
prob(*logic_program*)

Procedure 7: Q value of a logic program

5.6.1 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., [90, 25]). 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 [24] 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 [134], 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 SLD-resolution engine 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.

We would like to note that another system that looks at complete hypotheses instead of single clauses is TRACY [12].

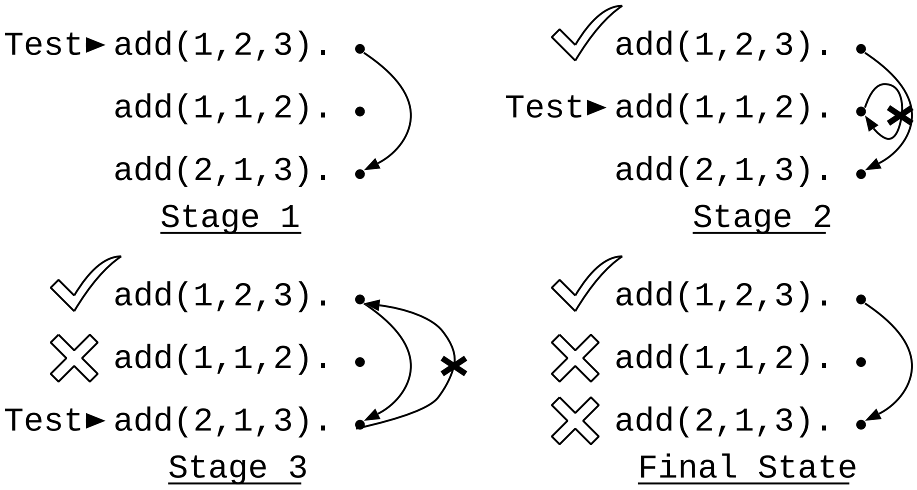


Figure 5.6: Example of estimating coverage of a individual recursive clause

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 5.6 illustrates this process for our example. At stage 1 the first instance $\mathrm{add}(1, 2, 3)$ is tested by resolving it with the recursive clause which uses the third instance $\mathrm{add}(2, 1, 3)$. 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 $\mathrm{add}(1, 1, 2)$ 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 $\mathrm{add}(2, 1, 3)$ would, when resolved, require the first instance $\mathrm{add}(1, 2, 3)$. 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 5.6 shows the stages LIME undertakens in this process.

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 SLD-resolution engine 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.

5.7 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 nonterminal 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 [133] used by LIME for calculating priors of logic programs is given in Table 5.1. The nonterminals LP, C, B, L, 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 5.2, where an element at the row corresponding to nonterminal X and the column corresponding to nonterminal 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 [18]. That is, the sum over all the sentences generated from this grammar is 1.

The calculation of $P(h)$ in LIME is trivial, as there is a unique derivation from the start nonterminal 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

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 5.1: Grammar use by LIME for calculating prior of hypothesis.

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

Table 5.2: Stochastic expectation matrix M for the logic program probabilistic context-free grammar.

$c\ P(h) > P(h \cup \{c\})$. 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.

5.8 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 4.1 that the θ value is a measure of the "proportion" of the instance space a concept covers. Formally, $\theta(C) = \sum_{x \in C} D_X(x)$, where D_X is the probability measure on the instance space X.

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 5.1. 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.

$$\theta = \sum_{e \in ext(h)} D_X(e) \tag{5.1}$$

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 $ext(h_1) \subset ext(h_2)$ 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.

Chapter 6

Empirical Results

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

Note, all the experiments are run on a *Pentium II Pro @ 200MHz* processor running the Solaris operating system. The aim of these experimental results is to show the superior performance of LIME in terms of accuracy. Although time complexity of learning is not one of the primary concerns of LIME, the run times are reasonable and given for comparison with other systems. The run times are given in seconds and are the sum of the user and system times. This provides a comparison of learning times and shows the feasibility of applying LIME to the domain in question.

6.1 Simple Tests

6.1.1 Boolean functions

These tests show LIME learning logic programs for simple Boolean functions — the results are shown in Table 6.1. The only background knowledge given are the predicates *true* and *false*. The noise parameter is also set to zero, as the training data is noise free and only consists of a few examples. LIME has no trouble learning these relations.

Name	Logic Program Learnt	$\lvert E^+ \rvert$	$\lvert E^- \rvert$	Time (Sec)
And	$and(A, B) \leftarrow true(A), true(B).$	1	3	0.17
Or	$or(A, B) \leftarrow true(A).$ $or(A, B) \leftarrow true(B).$	3	1	0.16
Xor	$xor(A, B) \leftarrow true(A), false(B).$ $xor(A, B) \leftarrow xor(B, A).$	2	2	0.15

Table 6.1: Learning simple boolean logic programs

6.1.2 Family relationships

A simple test domain commonly used to show the capability of ILP systems to learn relational concepts is that of family relationships. This domain highlights the advantages of first order systems over propositional ones, as the latter does not have the expressive power to represent

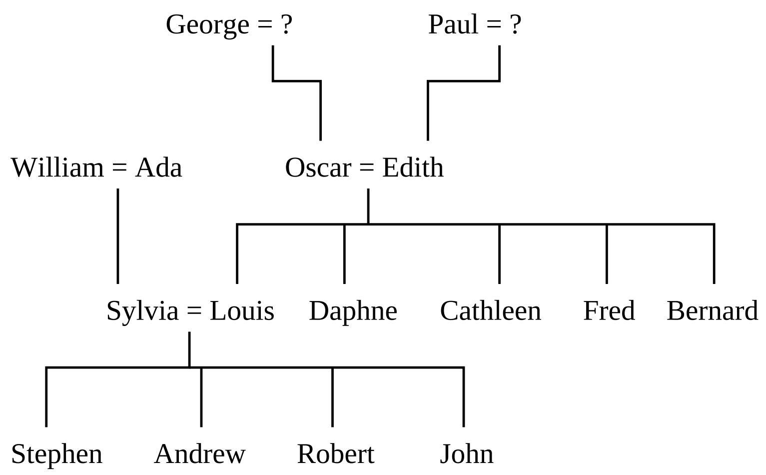


Figure 6.1: A family tree

concepts in this domain. The "family.pl" example data set provided with PROGOL [94] was used to generate these tests.

The family tree is shown in Figure 6.1.

We first show LIME's ability to learn the *parent* relation from only positive examples, where the background knowledge consists of the predicate *father* and *mother*. The logic program for *parent* consists of the following clauses:

$$parent(A, B) \leftarrow father(A, B).$$
$$parent(A, B) \leftarrow mother(A, B).$$

LIME learns the *parent* relation in 0.28 seconds from 11 positive examples and no negative example. This compares with PROGOL which takes 0.08 seconds from the same 11 positive examples.[1] The positive examples and a small portion of the background knowledge are shown in Table 6.2.

When *parent* relation is added to the background knowledge, both LIME and PROGOL are able to learn, from only positive examples, the *grandparent* and *grandfather* relations. All these tests take under a second for the learners to induce the result.

6.2 Recursive Logic Programs

6.2.1 Common recursive relations

Table 6.3 summarises experimental results on LIME's ability to learn a number of predicates that require recursive definitions. It should be noted that the data sets used in these experiments are noisy, except the *sort* relation, and not contiguous. Also, LIME's default parameters are used.

[1] LIME spends a lot of time in setting up various tables. As a result, its speed on trivial concepts with few examples is not as impressive as it is on nontrivial concepts.

Positive Examples	Background Knowledge
$parent(louis, stephen).$	$father(oscar, louis).$
$parent(oscar, daphne).$	$father(oscar, daphne).$
$parent(sylvia, robert).$	$father(louis, john).$
$parent(louis, andrew).$	$father(george, oscar).$
$parent(ada, sylvia).$	$father(paul, edith).$
$parent(oscar, bernard).$	$\vdots$
$parent(sylvia, john).$	$mother(sylvia, john).$
$parent(edith, fred).$	$mother(edith, louis).$
$parent(sylvia, stephen).$	$\vdots$
$parent(ada, sylvia).$	
$parent(william, sylvia).$	

Table 6.2: The positive examples and part of the background knowledge used when inducing the *parent* relation.

Logic Program	$\|E^+\|$	$\|E^-\|$	% incorrect[a]	Time (Sec)
$add(A,B,C) \leftarrow equal(A,C), zero(B).$ $add(A,B,C) \leftarrow inc(D,B), add(A,D,E), inc(E,C).$	16	10	3.8%	50
$member(A,B) \leftarrow components(B,A,C).$ $member(A,B) \leftarrow components(B,C,D), member(A,D).$	21	19	7.5%	80
$mult(A,B,C) \leftarrow inc(D,B), add(E,A,C), mult(A,D,E).$ $mult(A,B,C) \leftarrow zero(B), zero(C).$	16	24	2.5%	88
$sort(A,B) \leftarrow components(A,C,D), insert(C,E,B), sort(D,E).$ $sort(A,B) \leftarrow empty(A), empty(B).$	268	153	0.0%	94

[a]The column titled "incorrect" denotes the percentage of incorrectly classified examples in the training examples.

Table 6.3: Some recursive logic programs

6.2.2 Example logic programs form Bratko's textbook

A set of logic programs from Ivan Bratko's book *PROLOG Programming for Artificial Intelligence* [21] have been generated to assess ILP systems. These were obtained from the UCI Machine Learning Repository [71] 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 6.4 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, this is the only parameter set for LIME. FOIL by default allows negation as failure literals, we disable this option as all the concepts do not require hypotheses with negated literals. By disabling this option all three learners use a similar hypothesis space[2]. No parameters are set in PROGOL, although care is taken when constructing mode information required by the system. Table 6.5 gives the results of LIME, FOIL, and PROGOL on these data sets. Entries in this table are:

- *Successful* when the learner induces a hypothesis that exactly represents the concepts for any length lists and any set of constant symbols,
- *Restricted* success when the learner induces a hypothesis that is correct just for the training examples, or
- *Unsuccessful* if the learn returns a hypothesis which is incorrect on any of the training examples or a hypothesis that partially consists of a list of positive examples.

Note, that both LIME and FOIL successfully induces the target hypothesis for 10 out of the 16 logic programs, whereas PROGOL was successful on 5 out of the 16 logic programs.

6.2.3 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:

$$\begin{aligned}
qsort(A,B) &\leftarrow partition_left(A,C), partition_right(A,D),\\
&\quad\; concat(C,D,B), partition_right(B,D).\\
qsort(A,B) &\leftarrow partition_left(A,C), partition_right(A,D),\\
&\quad\; qsort(D,E), qsort(D,F), concat(E,F,B).
\end{aligned}$$

This requires extending the search of simple clauses. So the simple clause table size parameter is increased from the default 500 to 1600 and the noise parameter is set to 0. The

[2]This improves FOIL performance allowing it to correctly induce an extra 2 logic programs.

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

Table 6.4: Bratko's recursive logic programs

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

Table 6.5: Empirical results of LIME, FOIL, and PROGOL for recursive logic programs from Bratko's textbook.

background knowledge consists of the relations *partition_left*, *partition_right*, and *concat*. The relation *partition_left*(A, B) is true when B is equal to the list A with all the elements which are greater than or equal to the first element in A removed (e.g., *partition_left*$([3, 2, 6, 3, 1], [3, 6, 3])$. is true). Correspondingly, *partition_right*(A, B) is true when B is equal to the list A with all the element which are less than the first element in A removed (e.g., *partition_right*$([3, 2, 6, 3, 1], [2, 1])$. is true). These relations provide a simple way of partitioning a list — which is required for the *quick sort* relation. Note, the first clause in the induced hypothesis provides the base case of the logic program. LIME took 393 seconds to induce the above logic program.

6.3 Noise

We present results from three sets of experiments that compare LIME, FOIL and PROGOL at different levels of noise in the training data. The first experiment considers learning the simple nonrecursive *plus-two* relation. The second experiment considers learning the recursive predicate *add*. The third experiment is performed on the complex *krk* domain.

6.3.1 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:

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

In the above *inc* denotes the increment predicate available as background knowledge. A random selection of 30 positive and 30 negative examples is given to LIME. These examples include noise. The instance space is uniformally distributed over numbers ranging from 0 to 49. The predictive error of the induced hypothesis is measured against a noise-free test set that is generated by taking the 48 positive examples within the range and a random selection of 48 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 6.2. The error bars in the figure indicate

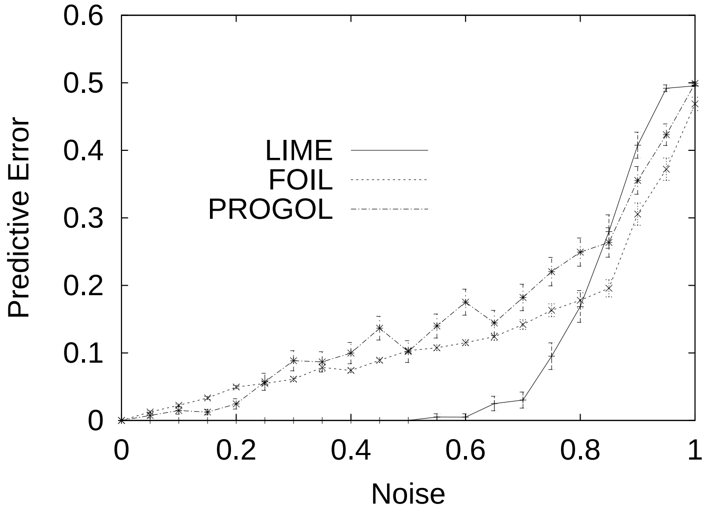


Figure 6.2: Predictive Error vs Noise for *plus2*

the standard error[3]. 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 [4].

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.

6.3.2 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:

$$add(A, B, C) \leftarrow equal(A, C), zero(B).$$
$$add(A, B, C) \leftarrow inc(D, B), add(A, D, E), inc(E, C).$$

Let the instance space $X = \{add(n_1, n_2, n_3) | n_1, n_2, n_3 \in \{0, 1, 2, \dots\}\}$. Let the measure D_X over X be defined by the distribution $D_X(add(n_1, n_2, n_3)) = g(n_1)g(n_2)g(n_3)$ where $g(n) = \frac{1}{4}(\frac{3}{4})^n$. 200 positive and 200 negative examples are generated using the process described in Section 4.1 (page 49). By performing 20 repetitions at each noise level an estimate in the induced hypothesis error is obtained. This estimate is calculated by generating a noise free test set consisting of all the instances with terms between 0 and 6, then calculating the percentage of positive examples for which the hypothesis is incorrect, correspondingly the percentage of negative incorrect examples is calculated, these two values are averaged to give the final estimate of error. Figure 6.3 shows the relationship between noise and error, comparing LIME with FOIL and PROGOL. The results show that the gap between LIME and these other systems widens further when the target concept

[3]The standard error provides a measure of the accuracy of the average. See Appendix A for more information.
[4]All our experiments are with FOIL, version 6.3 and with PROGOL, version 4.4.

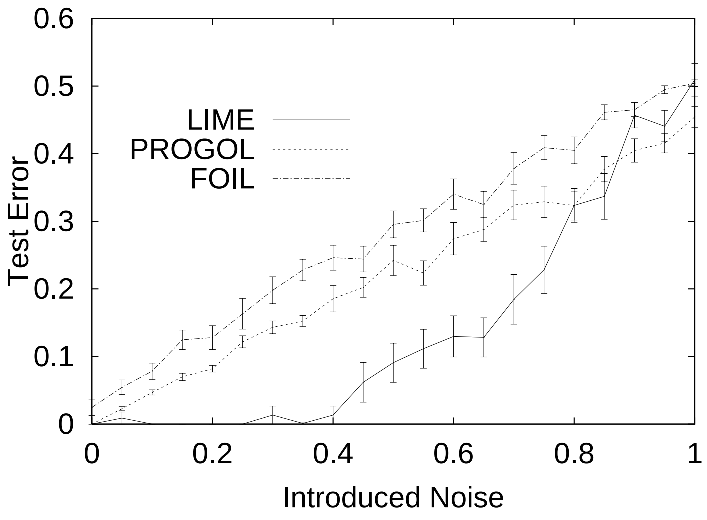


Figure 6.3: Error vs Noise for *add*

requires a recursive definition.

6.3.3 KRK domain

The KRK domain has been well studied, especially with respect to noise [92, 35]. 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 is noisy. Examples that are not noisy are chosen by randomly selecting using a uniform distribution the rank and file of each piece then determining if the state is illegal and labelling it appropriately. A noisy example is constructed by again randomly positioning each piece, then randomly labelling 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 10,000 random examples and calculating the proportion of these the hypothesis correctly classifies. Each trial is repeated 20 times and the mean and standard error of the accuracy is estimated.

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 6.4. The error bars show the standard error.

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.

6.4 Randomly Generated Domain

A randomly generated domain is created to examine how LIME performs as the target concept becomes more complex, or as the learner is faced with a more challenging task. The complexity of a target concept or the difficulty posed to the learner may be measured along a variety of

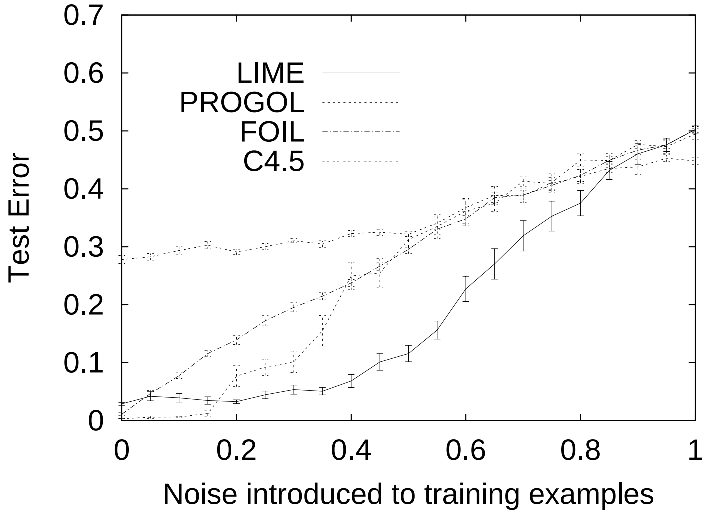


Figure 6.4: The predictive error as noise is introduced into the training examples in the KRK domain.

dimensions. These include: the number of clauses to express the concept, the number of literals that make up the clauses that express the concept, the amount of background knowledge given to the learner, and the amount of noise in the training set of examples. In this section we isolate each of these dimension and study them, thus forming a comparison between the ILP system investigated.

6.4.1 Number of clauses

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 a set number of clauses, 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 an estimate of the standard error of these averages is calculated. Figure 6.5 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 increases.

6.4.2 Number of literals in each clause

Another measure of complexity is the number of literals in each of the clauses. The same experimental set up is maintained, except this time the number of clauses in each of the randomly generated domains is fixed at 3, and the number of literals in each of the clauses is varied from 1 to 4. The results of these experiments are shown in Figure 6.6. It is interesting to note that PROGOL's performs increases from 1 to 2 literals in each of the clauses and then decreases from

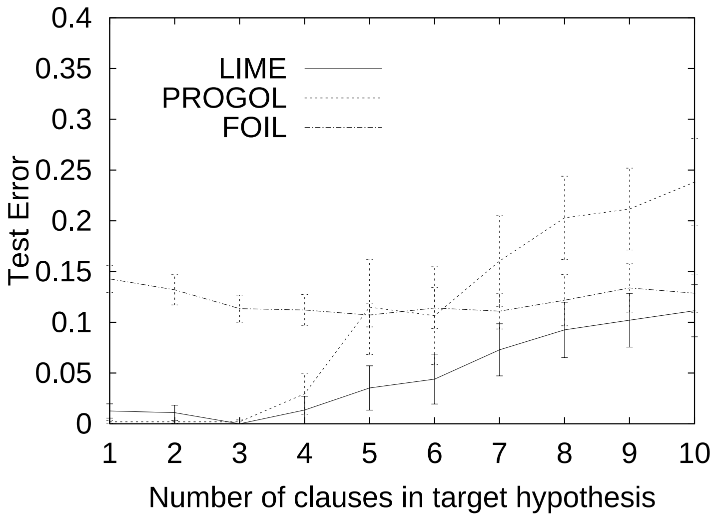


Figure 6.5: The predictive error vs number of clauses in the randomly generated domain.

2 to 4 literals. This suggests accuracy is not a linearly increasing relationship to the number of literals in each clause. LIME performs better than both FOIL and PROGOL on these tests.

6.4.3 Number of predicates in the background knowledge

As the number of predicates in the background knowledge increases, the hypothesis space becomes more expressive, thereby making the inductive process more difficult. By fixing the size of the target hypothesis and changing the number of predicates in the background knowledge, we investigate this aspect of complexity for the three systems. Once again we use the same experimental set up, however, in this experiment, we fix the number of clauses and length of each clause, then the number of randomly generated predicates in the background knowledge is varied from 1 to 10. The results of this experiment are shown in Figure 6.7. As expected the predictive error increases as the number of background predicates increase. This is seen clearly in both FOIL and PROGOL. Also, in LIME this increase takes effect more slowly than the other two system. Note, that at 100 predicates in the background knowledge LIME, FOIL, and PROGOL show average errors of 6.2% ,20.2%, and 44.7% [5] respectively.

6.4.4 Noise in the training set

The last of this series of experiments investigates predictive error as the level of noise in the training data is increased. Once again the same experimental setup is maintained with the target hypotheses having 3 clauses each with 3 literals. The noise is increased in steps of 5% and the results are shown in Figure 6.8. Note, the noise is classification noise where the class of a noisy example is flipped. It is generated by flipping the class of $\lceil n\epsilon \rceil$ examples in the training set, where n is the number of examples and ϵ is the level on noise.

[5] With standard errors of 2.2%, 2.8%, and 3.2% respectively.

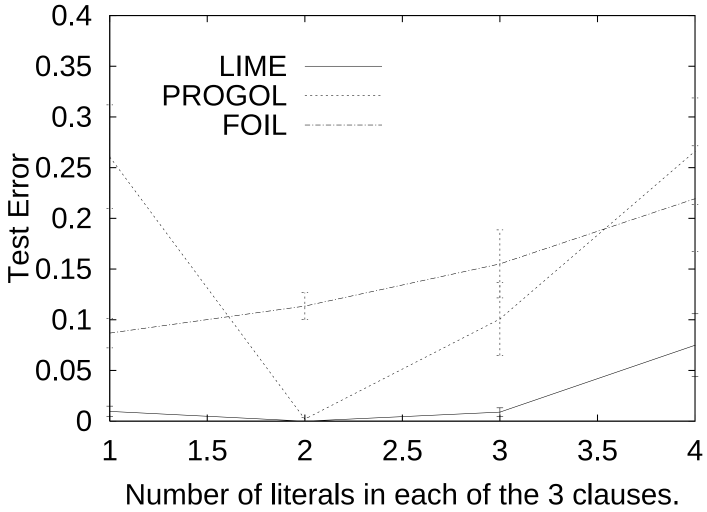


Figure 6.6: The predictive error vs number of literals in each of the 3 clauses in the randomly generated domain.

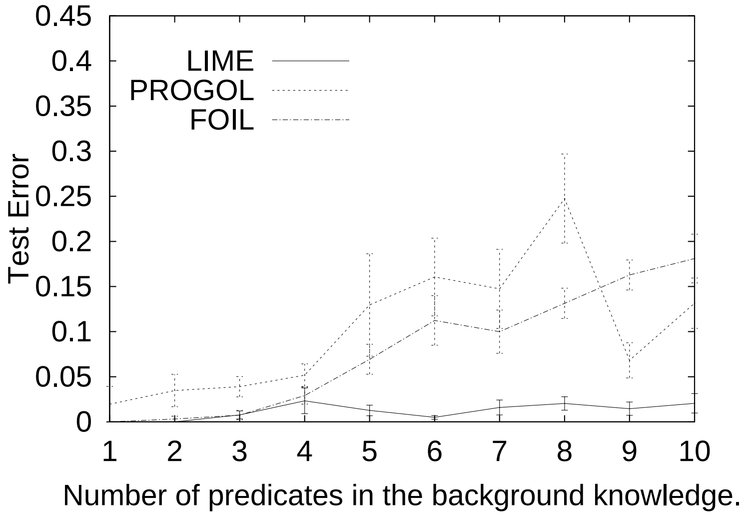


Figure 6.7: The predictive error vs number of predicates in the background knowledge in the randomly generated domain.

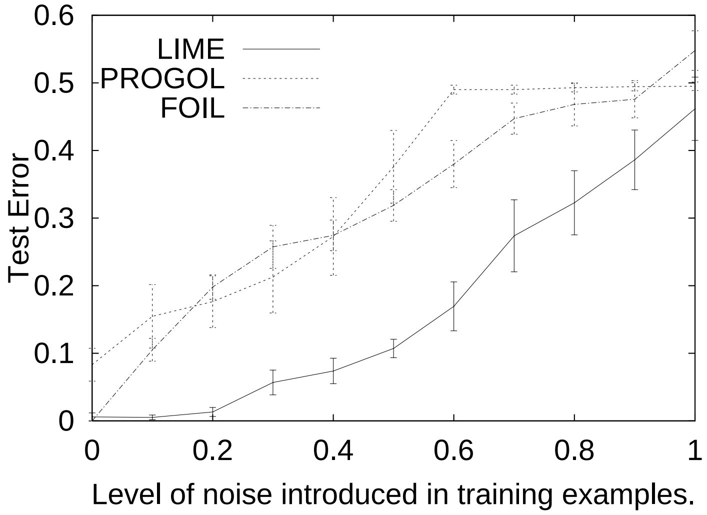


Figure 6.8: The predictive error vs noise introduced into the training examples in the randomly generated domain.

6.5 Learning from Positive Examples and from Negative Examples

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 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.

6.5.1 Plus two

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 $X = \{plus2(a_1, a_2) | a_1, a_2 \in \{1..n\}\}$ 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 1.92% 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 6.9. In all experiments the test set consists of 100 randomly selected positive examples and 100 randomly selected negative examples. The error bars show the standard error.

These experiments are also repeated with different levels of noise used in generating the training examples. The graphs for *plus2* and *notplus2* are shown in Figure 6.10.

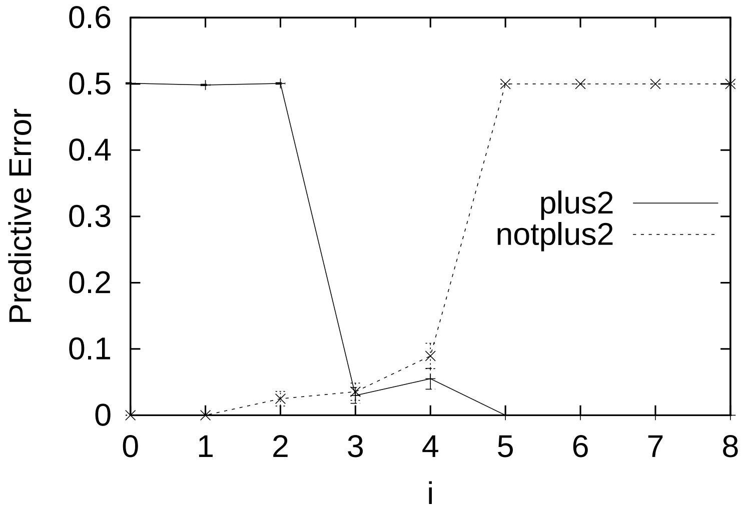


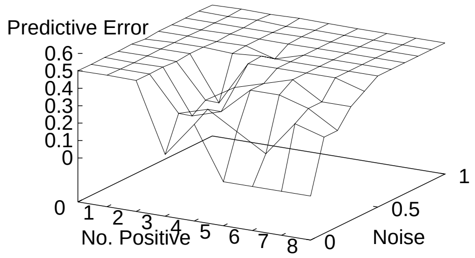
Figure 6.9: Error vs i (i = number of positive examples & 8-i = number of negative examples) for the *plus2* and *notplus2* logic programs. (e.g., At $i = 6$ LIME is provided 6 positive examples and 2 negative. This number of examples is sufficient(insufficient) for LIME to learn the *plus2*(*notplus2*) relation.)

6.5.2 Addition

In this set of experiments we examine the number of examples required by LIME to induce the *addition* relation from only positive examples. Note, as there are only positive examples the empty bodied clause is complete and consistent with respect to these examples. However, with enough positive examples the Q heuristic favours the *addition* relation over the empty body clause, as the θ estimate for *addition* is smaller, this outweighs the effect of the larger prior probability for the empty bodied 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 same distribution that is used in Section 6.3.2 of this chapter is used. The advantage of this approach is that 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 count the number of hypotheses that exactly expresses the *addition* relation. This is repeated 10 times and the mean and standard error are tabulated. This is repeated for different number of positive examples in increments of 10. The graph in Figure 6.11 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



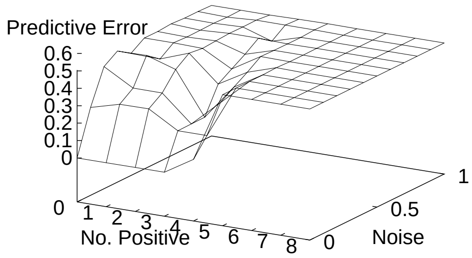
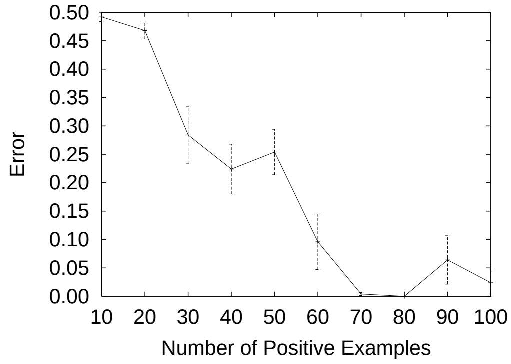


Figure 6.10: Error vs i (i = number of positive examples & 8-i = number of negative examples) for the *plus2* relation shown on the left and the *notplus2* relation shown on the right. These trials are conduced with different levels of noise used in generating the training examples.



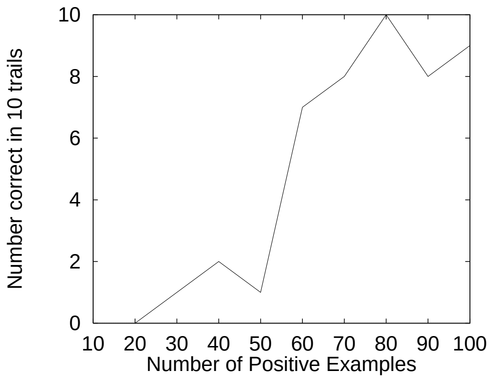


Figure 6.11: LIME inducing *addition* from only positive examples. The left graph shows error vs number of training examples. The right graph shows the number of times a correctly induced hypothesis is generated out of 10 trials vs number of training examples.

absent in negative only data, LIME will not learn recursive logic programs from only negative data.

6.6 Real World Data

6.6.1 Mesh resolution

The mesh domain involves determining the mesh resolution to use in analysing stresses in physical structures. As these structures may be described relationally a first order solution is appropriate. Dolšak applied GOLEM to this problem constructing a set of rules for determining mesh resolution [32, 33]. The same examples have been used to evaluate both FOIL and MFOIL [36]. For comparison we have used the same training and test examples. First, the finite mesh domain is described more thoroughly. Second, the example sets and background knowledge are described. Third, the results and an example rule induced by LIME are given and discussed.

One technique used by engineers to analyse stresses in physical structures is the finite element method. This involves representing the structure as meshes, then the deformation of each element may be computed using linear algebraic equations. One problem faced by engineers is determining the mesh size of each surface. If a surface is not complicated then a large mesh will accurately model the stress, however, if a surface is complicated or positioned at a critical location then a smaller mesh is required to accurately model the stresses. The more the number of mesh elements, the slower the computation of the linear algebraic equations. The objective of an engineer is to set the mesh size such that it is as large as possible while still accurately modelling the stresses. As aspects of each surface may be described relationally, the task of determining the mesh resolution may be represented in a first order language and, hence, forms a good real world test example for ILP.

The training examples have the form $mesh(E, N)$ where E is an edge of a structure and N is the number of elements used in determining the resolution. The examples consist of 5 structures which are labelled a, b, c, d, and e, these structures have 55, 42, 28, 57, and 96 edges respectively. For example $mesh(b10, 2)$ states that 2 is an appropriate number of elements for edge 10 of structure b. The background knowledge contains information that describes each edge and the relationship between other edges in the structure.

The experiment follows that described in Lavrač's and Džeroski's ILP book [61] which is a leave-one-out cross validation on the five structures. The results for FOIL, MFOIL, GOLEM, and LIME are shown in Table 6.6. Note, LIME performs better than both FOIL and GOLEM, although, worse than MFOIL. The predictive performance of all methods is poor.

6.6.2 Student loan

The student loan relational domain donated to the UCI Machine Learning Repository by Michael J. Pazzani consists of 1,000 examples. The target relation in this domain determines the students which are not required to repay the loan. The examples consist of 643 positive instances and 357 negative, and, the background knowledge contains a variety of relations including *sex*, *longest absence from school*, *enrolled*, *enlisted*, *unemployed*, *filed for bankruptcy*, *disabled*, *school*, *armed forces*, and *peace corps*. This domain has been previously studied in [96].

Structure	FOIL	MFOIL	GOLEM	LIME
A	17	22	17	21
B	5	12	9	9
C	7	9	5	6
D	0	6	11	15
E	5	10	10	3
Sum	34	59	52	54
Percentage	12	21	19	20

Table 6.6: Number and percentage of correctly classified mesh sizes by FOIL, MFOIL, GOLEM, and LIME for each of the test structures. Note, the result for FOIL, MFOIL, and GOLEM are given in [61].

Algorithm	Error	SE
LIME	3.6%	0.5
FOIL	2.4%	0.7
PROGOL	18.8%	0.7

Table 6.7: Results for LIME, FOIL, and PROGOL on the student loan data set.

A 10-fold cross validation was used to assess the predictive error of LIME, FOIL, and PROGOL on this data set. LIME produces an average error 3.6% with a standard error of 0.5. Whereas FOIL performs sightly better with an average error of 2.4% with a standard error of 0.7. PROGOL was had more difficulty inducing some of the key clauses producing an average error of 18.8% with a standard error of 0.7. These results are shown in Table 6.7. An example of a logic program produced by FOIL is given below:

$$
\begin{aligned}
nodue(A) \leftarrow\; & disabled(A). \\
nodue(A) \leftarrow\; & bankrupcy(A). \\
nodue(A) \leftarrow\; & unemployed(A). \\
nodue(A) \leftarrow\; & longest\_absense\_from\_school(A, B), lessequ5(B), \\
& enroled(A, C, D), greaterequ6(D). \\
nodue(A) \leftarrow\; & enlist(A, B), B <> nowhere, B <> fire\_department, \\
& B <> foreign\_legion.
\end{aligned}
$$

6.6.3 Tic-Tac-Toe

This database was also obtained from the UCI Machine Learning Repository and was created and donated by David W. Aha. It consists of the complete set of possible board configurations at the end of a tic-tac-toe game. The games in which "x" win are labelled positive and this forms the target relation. Note, that "x" always goes first and a board state is described with a nine element vector where each element describes the state of a square of the board. The possible states are "x", "o", or "b" where the square contains a cross, a nort, or the square is blank respectively. Crosses win when there are three "x" in line, this may occur in 8 possible ways. The database consists of 958 legal end game positions of which about 65.3% are positive, that is, wins for crosses.

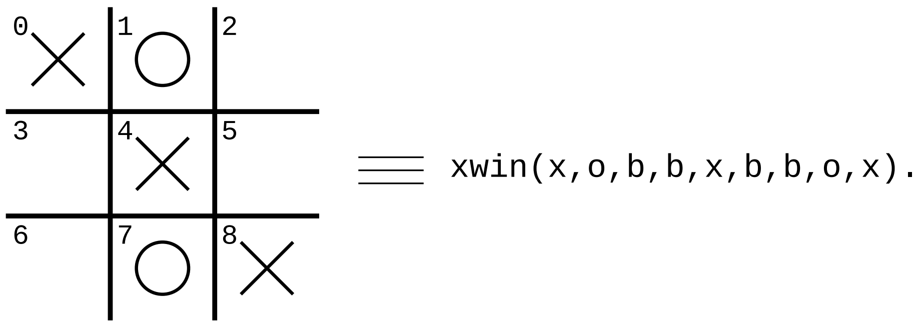


Figure 6.12: Board position with crosses winning and its representation in first order logic.

Algorithm	Accuracy
NewID	84.0%
CN2	98.1%
MBRtalk	88.4%
IB1	98.1%
IB3	82.0%
IB3-CI	99.1%

Table 6.8: Results reported for the tic-tac-toe domain in [4]

Figure 6.12 shows an example of a board end position where crosses wins and the corresponding representation in first order logic.

The experimental set up used in [4] has been followed so their results may be compared. The examples are randomly partitioned into training and test sets where training set consists of approximately 70% of the examples and the test set 30%. The learning system induces its hypothesis from the training set and the accuracy of this hypothesis is estimated using the test set. This entire process is repeated 10 times and the average accuracy is calculated.

The results reported by Aha [4] are shown in Table 6.8. Note that these are all propositional learners which find this concept more difficult to represent than with first-order learners. In fact, when Lime is given the background relation "three x's" it has no trouble inducing a hypothesis which exactly represents the concept in all 10 trials. Because the concept is simply represented with eight one literal clauses. Without this background knowledge, Lime has more difficulty. The target hypothesis consists of the same eight clauses, however, each clause now consists of three literals and not one. Lime is only able to exactly represent the concept 4 out of the 10 trials. These results are shown in Table 6.9.

This domain has also been studied in [67] and [66]. However, as the experimental set-up is

Threex in BG ?	Accuracy	SE
Yes	100%	0
No	96%	1.1

Table 6.9: Lime's results for the tic-tac-toe domain.

System	Error	SE
LIME	4.41%	0.76
LIME (-n 3)	3.38%	0.76
FFOIL	5.2%	-
C4.5	5.3%	-

Table 6.10: Result for breast cancer domain.

System	Error	SE
LIME (-n 3)	4.23%	0.27

Table 6.11: Result for breast cancer domain with 10 10-fold cross-validations

different the results are not comparable.

6.6.4 Wisconsin breast cancer

This domain is considered as it contains no relational information. Thus, we may directly compare LIME, a first order learner, and a propositional learner such as C4.5 [105]. The data consists of 699 examples with 9 attributes and two classes, benign which makes up 65.5% of the examples and malignant making up only 34.5%. Note, that malignant forms the target concept. A 10-fold cross-validation is conducted with this data and the averages are calculated, The results are found in Table 6.10. The results for FFOIL and C4.5 are reported in [106], note, the standard errors where not reported.

Interestingly, all three system give similar results, however, LIME performs a little better than the other systems. The same experiment was repeated using identical training examples, with the noise parameter set at 3% as opposed to the default 10%. This improved the average error of LIME by one percent. Unfortunately, between the averages of the trials with LIME, due to the small number of trials and the relatively high sample standard deviation, none of the means are statistically different. To determine if there is a statistically significant difference between them more trials have been performed.

Ten 10-fold cross-validations are performed on the same examples, where these examples are shuffled between each cross-validation, giving a total of 100 trials. The result is shown in Table 6.11. Note, the noise parameter is also set at 3% for these trials. Interestingly, there is no significant difference between them. This also confirms the usefulness of statistical tests as previously the average was lower with the -n 3 setting, however, the difference was not statistically convincing. Now, with more trials it is clear there is no significant difference between these two runs on the Wisconsin breast cancer examples.

6.6.5 Votes

The votes domain consists of the 1984 United States congressional voting records database. Each of the 435 instances contains 16 key votes identified by the congressional quarterly almanac where the congressman either voted *yea*, *nay*, or their vote is *unknown*. The classifying attribute is party affiliation of which 267 are democrats and 168 are republicans. This data set was obtained

System	Error	SE
LIME	4.31%	0.28

Table 6.12: Result in votes domain using 10 10-fold cross validation.

$$
\begin{aligned}
&add(A,B) \leftarrow isa\_D(B), split1(A,C), ise\_E(C).\\
&add(A,B) \leftarrow isa\_ED(B), split1(A,C), ise\_T(C).\\
&add(A,B) \leftarrow isa\_ED(B), split2(A,C), ise\_ER(C).\\
&\qquad\qquad\vdots\\
&add(A,B) \leftarrow isa\_IED(B), split1(A,C), ise\_Y(C).
\end{aligned}
$$

Table 6.13: A hypothesis induced in the past tense domain.

from the UCI Machine Learning Repository [71] and was donated by Jeff Schlimmer who collated the information from source documents [28]. Results of 90%-95% accuracy have previously been reported by Schlimmer [118].

Ten 10-fold cross validations were performed on the examples. The average and estimation of the standard error is shown in Table 6.12.

6.7 Past Tense of English Verbs

This domain consists of 1392 verbs and their past tense. The target concept is the mapping from verbs to its past tense. A portion of this mapping is given below:

$$
\begin{aligned}
&FORCE \rightarrow FORCED\\
&AFFORD \rightarrow AFFORDED\\
&SHOOT \rightarrow SHOT\\
&SCORE \rightarrow SCORED\\
&DASH \rightarrow DASHED
\end{aligned}
$$

This domain was initially used by connectionists to show they could model this interesting learning behaviour which was previously thought to require explicit rules [112]. Ling [64] strongly criticised this approach and presented more systematic results using a slightly modified version of C4.5. As ILP is a more appropriate representation for this domain Mooney and Califf [77, 78] learnt a first order decision list use FOIDL and were able to induce a compact, comprehensible, predictive hypothesis. Table 6.13 gives an example of a hypothesis induced by LIME in this domain.

Mooney and Califf's alphabetic past tense experiments are mimicked to compare LIME with FOIL and FOIDL. The 1396 examples used in these experiments were publicly provided by Ling [64]. The training positive examples are randomly selected from the example set without replacement, then the negative are generated from these examples. The test set consists of 500 randomly selected examples. These trials are repeated 10 times and an average and sample standard deviation is calculated to estimate the mean and standard error. This is repeated for 25,50,100,250, and 500 training examples. The results are shown in Figure 6.13. The same data is provided to FOIL and

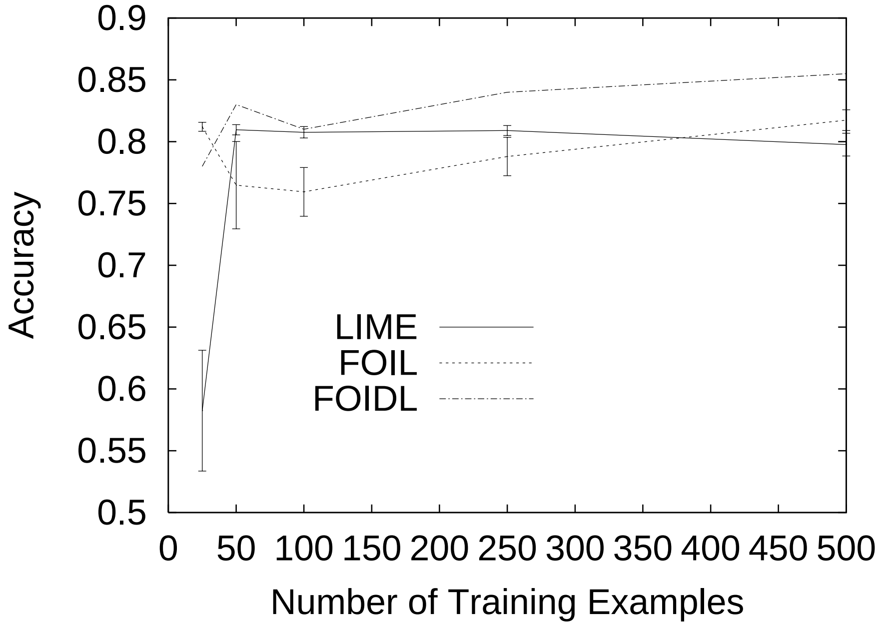


Figure 6.13: Accuracy verse number of training examples for the verb past tense data set.

the results are also graphed in Figure 6.13. The results for FOIDL are from Mooney and Califf's [78] paper. Figure 6.14 shows the number of clauses generated by both LIME and FOIL as the number of training examples increases.

6.7.1 Document understanding

The document understanding domain involves recognizing logical components of a document. This domain was introduced by Semeraro, Esposito, and Malerba [119, 38].

In particular the data consists of 30 letter documents, these letters are broken up into blocks. The background knowledge consists of attributes associated with blocks and the relationship between blocks within a letter. The logical components considered in these experiments are: *sender*, *receiver*, *reference number*, *date*, and *logo*. Other logical components such as signature and body are not considered by the learners. Each block many be one of the five logical components or may remain unlabelled. Figure 6.15 gives the page layout of letter $x1$ with blocks given labels from $x2$ to $x11$ this is expressed in the background knowledge as ground facts. Table 6.14 shows part of the background knowledge used in this domain.

We follow the experimental procedure used by Semeraro, Esposito, and Malerba [119] so LIME my be compared with the results reported in the conference proceedings regarding the FOCL system. The results are obtain by randomly partitioning the 30 letters into training and test sets where the training set consists of 20 letters and the test set consists of the remaining 10 letters. This is repeated 6 times with different partitions. Each of the five logical components are trained and tested separately. The positive (negative) examples consisted of all the blocks in the training letters which (don't) have the logical components label. The average error for each logical component is calculated by summing the number of errors of commission and omission for

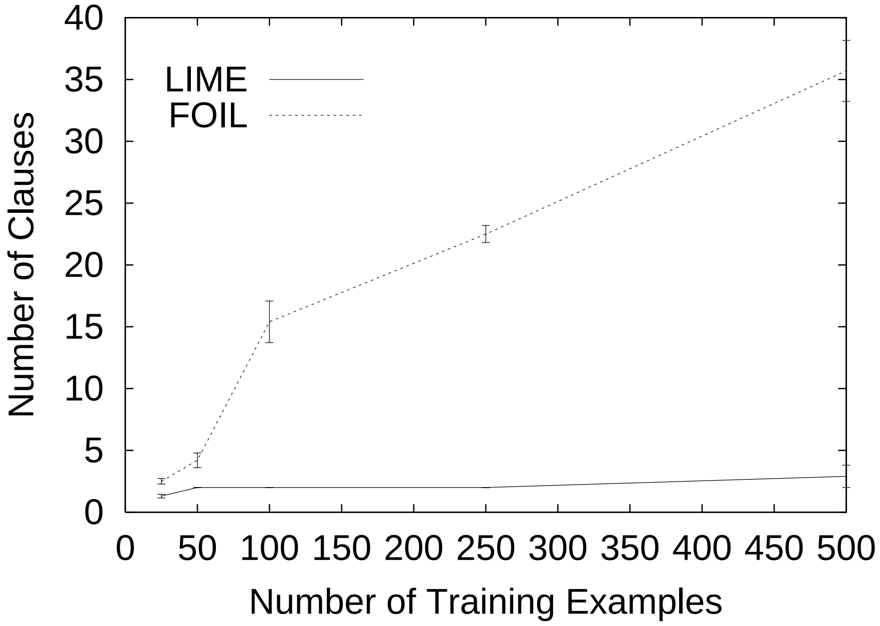


Figure 6.14: Number of clauses in induced hypothesis verse number of training examples for the verb past tense data set.

$width\_medium\text{-}small(x6).$
$height\_very\text{-}very\text{-}small(x6).$
$type\_text(x6).$
$contain(x1, x6).$
$position\_top\text{-}right(x6).$
$ontop(x6, x9).$
$align\_both\text{-}rows(x7, x6).$
$position\_top\text{-}horiz\text{-}band(x6).$
$position\_right\text{-}vert\text{-}band(x6).$
$aligned\_upper\text{-}row(x7, x6).$
$aligned\_middle\text{-}row(x7, x6).$
$aligned\_lower\text{-}row(x7, x6).$

Table 6.14: A portion of the background knowledge used in the document understanding domain.

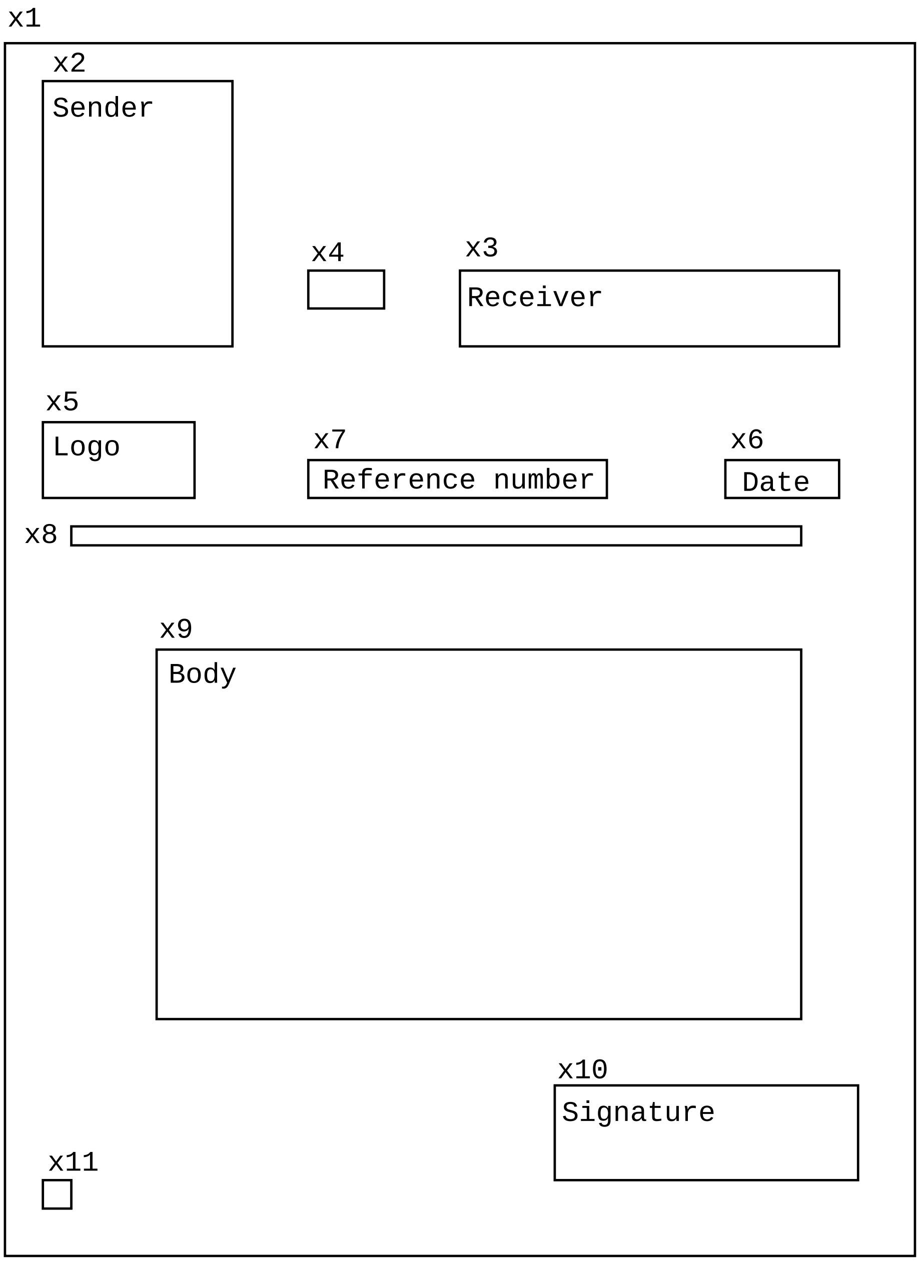


Figure 6.15: Page layout of the letter labelled *x1*.

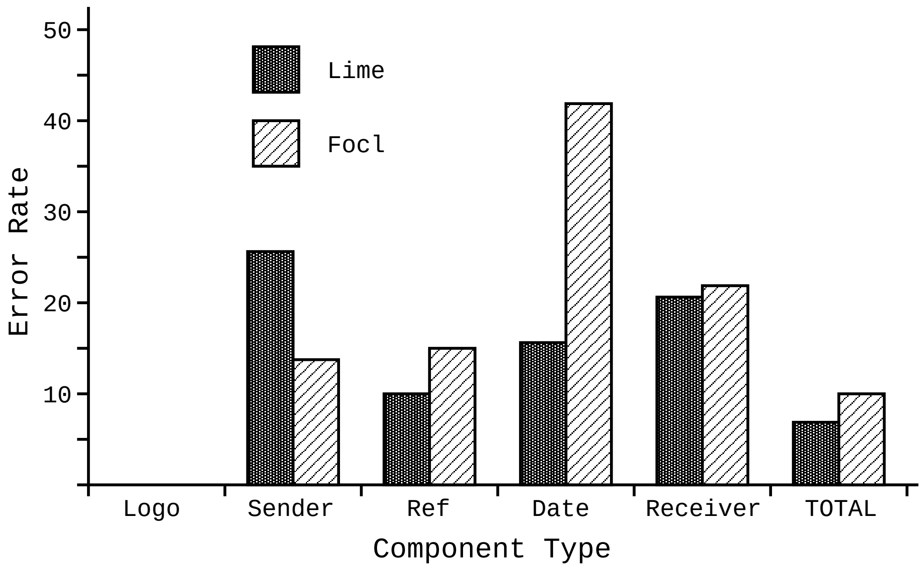


Figure 6.16: Comparison, between LIME and FOCL, of test result for each of the logical components considered.

the six trials divided by the sum of the components with the logical component label in question. The total average error is calculated by dividing the sum of errors of commission and omission for each logical component by the number of test blocks in the test data and then averaging these errors across the 6 trials. The test results are shown in Figure 6.16. Interestingly when LIME's parameters were altered to force it to induce larger hypotheses that match the training examples more exactly with an average error rate of 1% on the training examples. Then these hypotheses would over fit the data and perform poorly in terms of their predictive accuracy with an average error rate of 24% on the test examples.

6.7.2 Protein secondary structure

Tests were carried out on protein secondary structure prediction using GOLEM data from [91]. They report that initial application of GOLEM on these data sets produces a test set accuracy of 72%. These results are matched by LIME.[6] These initial results on real world data nicely complement LIME's effectiveness in noise handling and learning recursive predicates.

6.7.3 Mutagenesis

This set of experiments deal with the problem of discovering rules for mutagenicity in nitroaromatic compounds. Learning mutagenesis with ILP was first studied by Muggleton, Srinivasan, King, Sternberg [93, 124], who used PROGOL. They demonstrated a first order representation offers both predictive accuracy and a insight into the structural chemistry of mutagenesis.

[6]However, by iterating GOLEM [90], using the induced predicates as background knowledge, and modifying the clauses by hand, they have reported success rates of up to 81% accuracy on the test data.

Partition	“active”	“inactive”
Regression friendly	125	63
Regression unfriendly	13	29

Table 6.15: Number of molecules in each partition for the mutagenesis data set.

Learner	Regression friendly	Regression unfriendly
LIME (without lumo and log p)	0.87 (0.011)	0.79 (0.08)
LIME (with lumo and log p)	0.88 (0.035)	0.93 (0.06)
PROGOL (with lumo and log p)	0.88 (0.02)	0.83 (0.06)

Table 6.16: Results for the mutagenicity data set.

The experimental method of Muggleton, Srinivasan, King, and Sternberg [93] is followed. This allows us to compare result obtained. Note, the data sets are also provided by Muggleton et al.

The data set consists of 230 molecules which are partitioned into regression friendly and regression unfriendly sets. Molecules are labelled “active” when their mutagenic activity is greater than 0, as measured by the ames test, and “inactive” otherwise. The learner characterises what constitutes an “active” molecules. This characterisation may then be either used for prediction or to give insight into its structural chemistry. Table 6.15 shows the number of molecules either “active” or “inactive” in both of the partitions “regression friendly” and “regression unfriendly”.

Each molecule is described in the original background knowledge by a set of facts that describe each atom in the molecule and the bonds between atoms of the molecule.

$$\begin{array}{c} atm(d10, d10_1, c, 22, -0.119). \\ atm(d10, d10_2, c, 22, -0.119). \\ \vdots \\ atm(d10, d10_28, h, 3, 0.061). \end{array}$$

As LIME only induces determinate clauses, nondeterminate predicates are projected into determinate predicates.

The results of these trial are shown in Table 6.16.

A typical hypothesis induce by LIME is given below.

$$
\begin{aligned}
&active(0) \leftarrow atmh\text{-}c27(0), ring\text{-}size\text{-}5(0).\\
&active(0) \leftarrow phenanthrene(0).\\
&active(0) \leftarrow atmh\text{-}c29(0), ring\text{-}size\text{-}5(0).\\
&active(0) \leftarrow carbon\text{-}6\text{-}ring(0).\\
&active(0) \leftarrow atmc\text{-}c22\text{-}m142(0).\\
&active(0) \leftarrow atmc\text{-}h31\text{-}08(0).\\
&active(0) \leftarrow atmc\text{-}c22\text{-}m148(0).\\
&active(0) \leftarrow atmc\text{-}o40\text{-}m386(0).\\
&active(0) \leftarrow atmc\text{-}o40\text{-}m389(0).\\
&active(0) \leftarrow atmc\text{-}c22\text{-}m113(0).\\
&active(0) \leftarrow atmc\text{-}c21\text{-}m115(0).\\
&active(0) \leftarrow atmc\text{-}c29\text{-}10(0).\\
&active(0) \leftarrow atmh\text{-}o52(0).\\
&active(0) \leftarrow atmc\text{-}n38\text{-}817(0).\\
&active(0) \leftarrow atmc\text{-}h31\text{-}52(0).\\
&active(0) \leftarrow atmc\text{-}c22\text{-}m93(0).
\end{aligned}
$$

A typical hypothesis induce by LIME from the regression unfriendly data is given below.

$$
\begin{aligned}
&active(A) \leftarrow ring\text{-}size\text{-}5(A), lumo(A, B), lteqlum12(B).\\
&active(A) \leftarrow logp(A, B), lteqlp62(B), gteqlp56(B).\\
&active(A) \leftarrow atmh\text{-}n32(A), logp(A, B), lteqlp27(B).
\end{aligned}
$$

Chapter 7

Probabilistic identification results

The Q heuristic, which is presented in Chapter 4, lies at the heart of LIME. Chapter 6 empirically shows the effectiveness of LIME, especially with noisy training examples. However, this empirical study is limited as it only demonstrates the effectiveness of LIME on a particular set of domains. In this chapter we introduce an exact notion of probabilistic identification in the limit based on Laird [56, 57]. This yields a useful technique for assessing the effectiveness of a learner when training data is governed by a distribution and is possibly noisy. We then show that a learner which employs the Q heuristic will stochastically learn in the limit:

- any finite class of concepts, and
- a restricted infinite classes of concepts.

Of course, a finite class of concepts is trivially learnable from positive only data in the identification in the limit setting [48]. Hence, it is also learnable in the stochastic identification in the limit setting. What keeps our result from being trivial is the presence of noise in the data. Having presented this result, we explore conditions under which the result can be extended to an infinite class of concepts. The proof techniques for the infinite case builds on that of the finite case.

We show three classes of concepts, one of which is a restricted first order language, to be stochastically learnable in the limit using this approach. This forms an intermediate stage for being able to construct a hypothesis space for a general first-order language which can be shown to be stochastically learnable in the limit.

Finally, we show a more general result by modify the *Q-heuristic* and the learning algorithm using an approach similar to that of Laird [56]. This removes some of the previous restrictions. Also, this new result is more general than Laird's, as the assumptions made regarding the training data are more general.

7.1 Probabilistic Identification in the Limit

Probabilistic identification in the limit extends identification in the limit by replacing the teacher that presents all the examples to the learner with a teacher that uses a distribution to present examples to the learner. The criterion of success is correspondingly altered requiring that with probability 1 the learner induces a correct hypothesis all but finitely many times.

Generally, identification in the limit is concerned with characterising the class of concepts that may be learnt by any learner. However, we are more interested in assessing a particular learner, namely an inductive system using the Q heuristic. Hence, in this section we define probabilistic identification in the limit with this goal in mind.

Let X be the instance space and D_X be a probability measure over X. Note, D_X is a mapping from 2^X to $[0,1]$ and $D_X(\{x\})$ is simply written $D_X(x)$. We also assume X to be a countable set. Recall that members of 2^X are concepts. Let C be a class of concepts. The probability cover of a concept c, defined $\theta(c)$, is $D_X(c) = \sum_{x \in c} D_X(x)$.

The error or difference between two concepts c_1 and c_2 with respect to the probability measure D_X is defined as $\mathrm{error}(c_1, c_2) = \theta(c_1 \Delta c_2)$. By using *error* to evaluate a hypothesis the hypothesis only needs to be correct on instances which have nonzero probability in the instance space distribution. This is reasonable as the learner will never be presented with an instance with zero probability.

We let $\mathbb{E} = X \times \{\mathrm{Pos}, \mathrm{Neg}\}$ be the set of all labelled instances of the instance space X. We usually refer to labelled instances as examples. An example text $E \in \mathbb{E}^\infty$ is an infinite sequence of examples. The learner conjectures a hypothesis from an initial finite sequence of E. This initial finite sequence of E of length m is denoted $E[m]$. We let SEQ denote the set of all initial finite sequences, $\{E[m] | E \in \mathbb{E}^\infty \wedge m \in \mathbb{N}\}$.

Let h be a hypothesis. In the present work, h is a computer program. A hypothesis space is a sequence (usually infinite) of hypothesis. We assume that the hypothesis space H under consideration is enumerable. Let $h_0, h_1, \ldots$ be an enumeration of H. We further assume that H is uniformly decidable, i.e., there exists a computable function $f : \mathbb{N} \times X \to \{0,1\}$ defined below:

$$f(i,x) = \begin{cases} 1 & \text{if } x \in \mathit{ext}(h_i), \\ 0 & \text{otherwise.} \end{cases}$$

We say that a hypothesis space H is complete with respect to a concept class C if for each $c \in C$, there is a hypothesis h in the space H such that $c = \mathit{ext}(h)$.

We define a learner M to be a computable machine that implements a mapping from SEQ into H.

We also assume the learner is able to compute $\theta(\mathit{ext}(h))$ for any h in the hypothesis space H. Note that such a capability is unlikely to be available to any computable learner, however, $\theta(\mathit{ext}(h))$ may always be estimated and its exact value is not critical to induce the hypothesis with the largest Q-value.

Definition 7.1.1 (Convergence) *Learner M converges to hypothesis h on E just in case for all but finitely many $m \in \mathbb{N}$, $M(E[m]) = h$. This is denoted $M(E)\downarrow = h$.*

A stochastic process GEN is used to generate these example texts. This process may be formulated in a variety of ways depending on the kind of tests against which the learner is to be benchmarked. The example texts generated will reflect the target concept, although it may not be an exact or complete representation of the target concept as it may contain spurious or missing examples.

We now introduce a general stochastic process for generating example texts, this process is denoted $\mathrm{GEN}^{O}_{\langle \mu_p, \mu_n \rangle}$. The parameters $\langle \mu_p, \mu_n \rangle$ governs the amount of noise in the texts generated. μ_p gives the level of noise in the positive examples and correspondingly μ_n for the negative examples. In most cases $\mu_p = \mu_n$, however, it is useful to allow these parameters to be different in some cases. By setting $\mu_p = \mu_n = 0$ the process will generate noise free example texts. The parameter $O \in \{\mathrm{Pos}, \mathrm{Neg}\}^{\infty}$ is an oracle which determines which elements will be positive and negative in the sequence generated by $\mathrm{GEN}^{O}_{\langle \mu_p, \mu_n \rangle}$ prior to any instance being selected. The n'th element in the oracle O is denoted $O(n)$. By using an oracle we may model a variety of situations. For example, the oracle may determine all examples in the example text to be negative, hence we will model learning from only negative examples. We show the stochastic convergence results for any oracle, thus proving the result for a variety of situations. We may also place a probability measure over $\{\mathrm{Pos}, \mathrm{Neg}\}^{\infty}$ and assume O is stochastically generated by such a measure. As the stochastic learning result is shown for any $O \in \{\mathrm{Pos}, \mathrm{Neg}\}^{\infty}$ the result will be also true for an oracle generated by any stochastic process.

Algorithm 11 describes $\mathrm{GEN}^{O}_{\langle \mu_p, \mu_n \rangle}(c, X, D_X)$. In each cycle of the main loop the next example in the example text is generated. The oracle O is used to determine if the next example will be positive or negative. If the oracle decides that the next example will be positive, the following process is used: a biased coin is flipped where the probability of the coin coming up "Heads" is μ_p and "Tails" is $1 - \mu_p$; if the coin comes up "Heads" then any instance is randomly selected from X using D_X and output as a positive example, if the coin is "Tails" then any instance is randomly selected from c using the distribution J^c_X where:

$$J^c_X(x) = \begin{cases} D_X(x)/\theta(c) & \text{if } x \in c, \\ 0 & \text{otherwise.} \end{cases}$$

A similar process is used if the oracle decides that the next example will be negative. This algorithm generates a text which reflects the concept c, where the sign of each example in the text matches the sign of the corresponding element in O and the parameters $\langle \mu_p, \mu_n \rangle$ determine the levels of noise introduced into the example text.

We now calculate the probability measure over $\mathbb{E}$ for each example generated by $\mathrm{GEN}^{O}_{\langle \mu_p, \mu_n \rangle}$. There are two possible probability measures an example may have, either G^+ or G^-. The n'th element of the example text will have probability measure G^+ if $O(n) = \mathrm{Pos}$, otherwise it will have probability measure G^- when $O(n) = \mathrm{Neg}$.

So when the oracle O determines the n'th example to be labelled "Pos", that is $O(n) = \mathrm{Pos}$, example $\langle x, s \rangle$ is governed by:

$$G^+(\langle x, s \rangle) = \begin{cases} \mu_p D_X(x) + (1 - \mu_p)(J^c_X(x)) & \text{if } s = \mathrm{Pos}, \\ 0 & \text{if } s = \mathrm{Neg}. \end{cases}$$

Correspondingly, for the examples where $O(n) = \mathrm{Neg}$:

$$G^-(\langle x, s \rangle) = \begin{cases} \mu_n D_X(x) + (1 - \mu_n)(J^{\overline{c}}_X(x)) & \text{if } s = \mathrm{Neg}, \\ 0 & \text{if } s = \mathrm{Pos}. \end{cases}$$

Input :

A target concept c and probability measure D_X over subsets of the instance space X.

The noise levels $\mu_p, \mu_n \in [0, 1)$. An oracle O for deciding the class label for each example in the text.

Output :

An example text $E = \{e_1, e_2, \dots\}$.

$n := 0$

repeat

next_example_class := $O(n)$

if next_example_class = Pos **then** /* Generate a positive example. */

  coin := flip_bias_coin(μ_p)

  if coin = Head **then**

    x := random_select(D_X)

  else

    x := random_select(D_X)

    while $x \notin c$ **do**

      x := random_select(D_X)

    od

  fi

  output $e_n = \langle x, \mathrm{Pos}\rangle$

else /* Generate a negative example. */

  coin := flip_bias_coin(μ_n)

  if coin = Head **then**

    x := random_select(D_X)

  else

    x := random_select(D_X)

    while $x \in c$ **do**

      x := random_select(D_X)

    od

  fi

  output $e_n = \langle x, \mathrm{Neg}\rangle$

fi

$n := n + 1$

forever

Algorithm 11: $\mathrm{GEN}^{O}_{\langle \mu_p, \mu_n \rangle}(c, X, D_X)$

As D_X, $J_X^{\overline{c}}$, and J_X^c are probability measures on X it is straightforward to show that G^+ and G^- are probability measures on $\mathbb{E}$.

Note $G^+(\mathbb{E}) = 1$ and $G^-(\mathbb{E}) = 1$. These measures are used to define the probability measure $\text{Prob}_{\text{GEN}^O_{\langle \mu_p, \mu_n \rangle}(c,X,D_X)}$ on the σ-field $\mathcal{F} \subset 2^{(\mathbb{E}^\infty)}$, where $\mathcal{F}$ is the σ-field generated from the prefix sets of $2^{(\mathbb{E}^\infty)}$. Note that for every prefix set $B_\sigma = \{E \in \mathbb{E}^\infty | \sigma = E[|\sigma|]\}$ where $\sigma = \langle e_0, e_1, \ldots, e_n \rangle$ we have $\text{Prob}_{\text{GEN}^O_{\langle \mu_p, \mu_n \rangle}(c,X,D_X)}(B_\sigma) = \prod_{n<|\sigma|} f(e_n, O(n))$, where

$$f(\langle x, s \rangle, o) = \begin{cases} G^+(\langle x, s \rangle) & \text{if } o = \text{Pos}, \\ G^-(\langle x, s \rangle) & \text{if } o = \text{Neg}. \end{cases}$$

Note that for any $A \in \mathcal{F}$ their exists a $\{B_i\}_{i=1}^\infty$ such that B_i's are mutually disjoint, the B_i's are elements of the prefix set, and we have $A = \cup_{i=1}^\infty B_i$. Let $\text{Prob}_{\text{GEN}^O_{\langle \mu_p, \mu_n \rangle}(c,X,D_X)}(A) = \sum_{i=1}^\infty \text{Prob}_{\text{GEN}^O_{\langle \mu_p, \mu_n \rangle}(c,X,D_X)}(B_i)$. Now it is possible to show $\text{Prob}_{\text{GEN}^O_{\langle \mu_p, \mu_n \rangle}(c,X,D_X)}$ is a measure over the σ-field $\mathcal{F}$ and we have:

$$\text{Prob}_{\text{GEN}^O_{\langle \mu_p, \mu_n \rangle}(c,X,D_X)}\left\{ \mathbb{E}^\infty \right\} = 1$$

We refer the reader to *Measure Theory and Probability* by Adam and Guillemin [1] or *Probability and Measure* by Billingsley [15] for further information on measure theory.

Using $\text{GEN}^O_{\langle \mu_p, \mu_n \rangle}$ provides a flexible way of modelling different forms of training data. We now provide a list of common models for training data and show how these are specializations of $\text{GEN}^O_{\langle \mu_p, \mu_n \rangle}$.

Noise free, positive examples: If we set $\mu_p = \mu_n = 0$ and set $O = \langle \text{Pos}, \text{Pos}, \text{Pos}, \ldots \rangle$ the training data will be noise free and positive. The distribution of this training data will reflect a normalized version of the instance space distribution, where elements outside the target concept have probability zero of appearing in a text. This is identical to the assumption about the training data used by Montagna and Simi [76] who showed that whatever may be learnt in the limit from both positive and negative data may also be stochastically learnt in the limit from only positive data. This result assumes D_X is approximately computable. This is also similar to the model used by Angluin [5] when she considered TXTEX-identification. Angluin allows a null or empty element, denoted $\star$, to be part of the text, to facilitate modelling a text for the empty language.

Noisy, independently generated examples: Laird's [56, 57] classification noise process assumes that instances are chosen according to some distribution and then correctly labelled according to the target concept. After this a demon with probability ξ flips the class label from positive to negative or from negative to positive, thereby creating noise in the training data[1]. This process generates an example text where each example is independent and has the following

[1] Laird [56] uses μ for the noise parameter, however, as it different to the noise parameter used here, we use ξ to refer to Laird's noise parameter.

distribution:

$$P_{\text{Laird}}(\langle x, s\rangle) = \begin{cases} (1-\xi)D_X(x) & \text{if } s = \text{Pos} \wedge x \in c, \\ \xi D_X(x) & \text{if } s = \text{Pos} \wedge x \notin c, \\ \xi D_X(x) & \text{if } s = \text{Neg} \wedge x \in c, \\ (1-\xi)D_X(x) & \text{if } s = \text{Neg} \wedge x \notin c. \end{cases}$$

Now let us see how this distribution can be modelled in our framework. We now place a probability measure over $\{\text{Pos}, \text{Neg}\}^\infty$ such that each element is the sequence is independent and is "Pos" with probability w and "Neg" with probability $1-w$. We denote an oracle produced by such a distribution O_w.

Now, each element in the example text produced by $\text{GEN}^{O_w}_{\langle \mu_p, \mu_n \rangle}$ will be independent and have the following distribution:

$$P(\langle x, s\rangle) = \begin{cases} w(\mu_p + (1-\mu_p)/\theta(c))D_X(x) & \text{if } s = \text{Pos} \wedge x \in c, \\ w\mu_p D_X(x) & \text{if } s = \text{Pos} \wedge x \notin c, \\ (1-w)\mu_n D_X(x) & \text{if } s = \text{Neg} \wedge x \in c, \\ (1-w)(\mu_n + (1-\mu_n)/(1-\theta(c)))D_X(x) & \text{if } s = \text{Neg} \wedge x \notin c. \end{cases}$$

Now, let $w = \theta(c) + \xi - 2\theta(c)\xi$, $\mu_p = \frac{\xi}{\theta(c)+\xi-2\theta(c)\xi}$, and $\mu_n = \frac{\xi}{1-\theta(c)-\xi+2\theta(c)\xi}$. Then the distribution for each example in the example text generated by $\text{GEN}^{O_w}_{\langle \mu_p, \mu_n \rangle}$ will be identical to P_{Laird}. It follows, their probability measures over $\mathbb{E}^\infty$ will also be identical. Hence, by showing a result for stochastic learning with $\text{GEN}^{O}_{\langle \mu_p, \mu_n \rangle}$ we correspondingly show the result for Laird's model of training data.

Noise free, with both positive and negative examples: Learning with both positive and negative examples is the same as EX-identification where the functions in question have range restricted to either "Pos" or "Neg". Angluin [5] when considering EX-identification in a probabilistic setting assumes that each example is independent in the text and the probability of an example appearing is based on a distribution from the range of the function. This gives us the following distribution over the examples:

$$P(\langle x, s\rangle) = \begin{cases} D_X(x) & \text{if } s = \text{Pos} \wedge x \in c, \\ 0 & \text{if } s = \text{Pos} \wedge x \notin c, \\ 0 & \text{if } s = \text{Neg} \wedge x \in c, \\ D_X(x) & \text{if } s = \text{Neg} \wedge x \notin c. \end{cases}$$

If the oracle O_w, as defined in the previous model, where $w = \theta(c)$ and $\mu_p = \mu_n = 0$, then $\text{GEN}^{O_w}_{\langle \mu_p, \mu_n \rangle}$ gives the same distribution over each of the generated examples in the text.

By showing a learner to stochastically identify a class of concepts C when examples are provided by $\text{GEN}^{0}_{\langle \mu_p, \mu_n \rangle}$, we also show that the learner will stochastically identify C when examples are provided by distributions used in the other models.

Definition 7.1.2, of probabilistic identification in the limit, is based on the definition given in Laird's thesis [56, 57].

Definition 7.1.2 (Probabilistic identification in the limit) *Given an instance space X and a probability measure D_X over X. A learner M is said to identify the class of concepts C stochastically in the limit, with respect to a hypothesis space H, if and only if*

(a) *examples are provided by GEN, and*

(b) $(\forall c \in C)\ Prob_{GEN(c,X,D_X)} \left\{ E \in \mathbb{E}^\infty \middle| M(E)\downarrow = h \wedge error(ext(h), c) = 0 \right\} = 1.$

This setting has the expected property that any subset of a class that is stochastically learnable in the limit is also stochastically learnable in the limit with respect to the same hypothesis space. This proposition is formally stated below, the proof of which is omitted as it follows trivially from the above definition.

Proposition 7.1.1 *Given an instance space X and a probability measure D_X over X. Let H be a hypothesis space. Let C, C' be concept classes where $C' \subseteq C$. If C is stochastically learnable in the limit with respect to H, then C' is also stochastically learnable in the limit with respect to H.*

Laird [56] shows that any class of concepts that has a recursively enumerable set of hypotheses may be stochastically identified in the limit. This assumes both positive and negative examples are presented to the learner according to the distribution. This result is then extended by Laird to include noise in the training examples. Two central results, used in the proofs by Laird, are also central to the results given in this chapter.

First, the Borel-Cantelli lemma[2] which states that given any infinite sequence of events, if the sum of their probabilities is finite then with probability 1 only a finite number of events will occur. This is very useful as we may consider an event to be "the learner inducing an incorrect hypothesis in round m." An upper bound on the probability of this event may be calculated. Now, by showing that the sum of these bounds over all $m \in \mathbb{N}$ is finite, we prove that the learner, with probability 1, will induce a correct hypothesis all but finitely many times.

Lemma 7.1.1 (Borel-Cantelli) *Let $A_1, A_2, \ldots$ be an infinite sequence of events. If $\sum_{n=1}^{\infty} P(A_n)$ is finite, then the probability that infinitely many distinct A_n's occur is 0.*

Second, Hoeffding's probability inequality provides a bound on the probability of sums on bounded random variables being lower than a fixed value [45]. This is used to bound the probability of inducing an incorrect hypothesis.

Lemma 7.1.2 (Hoeffding) *Let sequences $(a_i)_{1 \leq i \leq m}$ and $(b_i)_{1 \leq i \leq m}$ be such that for all $1 \leq i \leq m$, $a_i \leq b_i$. Let $Z_1, Z_2, \cdots Z_m$ be independent random variables, $S = Z_1 + \cdots + Z_m$, $\overline{Z} = S/m$, and $E(\overline{Z}) = E(S)/m$, where $a_i \leq Z_i \leq b_i$ for all $1 \leq i \leq m$. Then for $t > 0$:*

$$Pr\{\overline{Z} - E(\overline{Z}) \leq -t\} \leq e^{\left(\frac{-2m^2t^2}{\sum_{i=1}^{m}(b_i - a_i)^2}\right)}$$

[2]The reader is directed to an introductory text on measure theory such as, *Measure Theory and Probability* [1] for more information.

7.2 Probabilistic Identification with the Q-heuristic

Let m be the total number of examples presented to the learner, so $m = n + p$, where p is the number of positive examples and n is the number of negative examples. Let $\mathrm{GEN}^{O}_{\langle \mu_p, \mu_n \rangle}$ generate the example text E. The learner M, given initial sequence $E[m]$ induces the hypothesis $M(E[m])$.

The order of presentation of examples, the sign of examples, and the proportion of positive and negative examples is dependent on the choice of oracle O. Since these aspects of the example presentation are not crucial for the learning algorithm, we assume that the learner is provided with a multiset of positive examples (of cardinality p) a multiset of negative examples (of cardinality n).

The algorithm simply works by choosing the hypothesis with the maximum[3] Q value given the current examples. In general there may be a set of hypotheses with equal Q values. To stop the algorithm alternating between them, the hypothesis with the minimum index[4] is chosen. If this minimum index selection is removed then the algorithm will still learn stochastically in the limit, although only in the behaviourally correct sense. Note that the algorithm is computable as it only must consider a finite initial portion of the possibly infinite hypothesis space. This follows from the bound obtained in proposition 4.3.2 [page 53].

Input :

An indexed hypothesis space H.
A prior probability distribution over H.
A function θ for evaluating the theta value of a hypothesis.
A sequence $\sigma = E[m]$ of m examples from the example text E.
The noise parameter $\epsilon \in [0, 1)$ such that $\mu_p \leq \epsilon$ and $\mu_n \leq \epsilon$.

Output :

A hypothesis h.

$h := \mathrm{minindex}(\mathrm{argmax}_{h \in H} Q_\sigma(h))$
output h

Algorithm 12: Stochastic Identification using the Q heuristic

Several lemmas are required for the following proofs. These lemmas are placed at the end of the chapter so the reader is not distracted from the main flow of the proofs.

7.2.1 A finite concept class

Theorem 7.2.1 *Let C be any finite concept class and let H be any hypothesis space which is complete for C. Then for any noise parameter ϵ, there exists a learning algorithm that stochastically identifies C in the limit with respect to H when examples are provided by $GEN^{O}_{\langle \mu_p, \mu_n \rangle}$ for any oracle O and any $\mu_p \leq \epsilon$ and $\mu_n \leq \epsilon$.*

PROOF:

[3] The notation $\mathrm{argmax}_{h \in H} Q(h)$ denotes the set $\{h \in H \mid (\forall h' \in H)\ Q(h) \geq Q(h')\}$.
[4] The notation $\mathrm{minindex}(S)$ denotes the hypothesis $h \in S$ such that $(\forall h' \in S - \{h\})\ h < h'$, where $<$ is a total ordering on H.

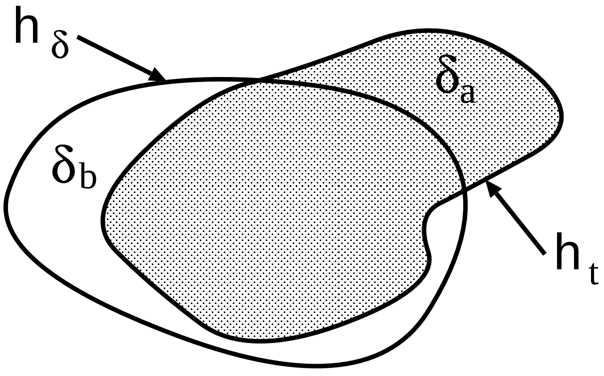


Figure 7.1: The cover of h_t and h_δ.

The proof compares h_t, a hypothesis that correctly classifies the target concept, with h_δ, a hypothesis that is in error. The value of $Q(h_t) - Q(h_\delta)$ is partitioned into three parts: a fixed constant, a sum of a list of random variables each corresponding to a positive example, and a sum of a list of random variables each corresponding to a negative example. The expected value for each of these random variables is shown to be positive. Assuming that the sum of these random variable is at least half the expected sum, we will have $Q(h_t) > Q(h_\delta)$ at some point, even when the fixed constant is negative. Applying Hoeffding's inequality, we compute a bound on the failure of this assumption. This bound is then used in conjunction with the Borel-Cantelli lemma to show that the class of concepts can be stochastically identified in the limit.

Suppose the learner is confronted with the concept $c \in C$ and there is a hypothesis, say h_t, such that $\mathrm{error}(c, \mathit{ext}(h_t)) = 0$. Suppose also that there is a hypothesis h_δ such that $\mathrm{error}(\mathit{ext}(h_t), \mathit{ext}(h_\delta)) = \delta$, where δ may be broken down into two parts $\delta_a = \theta(\mathit{ext}(h_t) - \mathit{ext}(h_\delta))$ and $\delta_b = \theta(\mathit{ext}(h_\delta) - \mathit{ext}(h_t))$. This is visually shown by the Venn diagram in Figure 7.1. Note, $\delta = \delta_a + \delta_b$.

Consider the difference between the Q value of the correct hypothesis h_t and a hypothesis which is in error, h_δ. Let $D = Q(h_t) - Q(h_\delta)$. If D is positive then $Q(h_t) > Q(h_\delta)$ and hence the correct hypothesis h_t is selected over h_δ. Recall (Chapter 4) that Q is defined as:

$$\begin{aligned} Q(h) = &\lg(P(h)) + |\mathrm{TP}_h| \lg\left(\frac{1-\epsilon}{\theta(\mathit{ext}(h))} + \epsilon\right) + \\ &|\mathrm{TN}_h| \lg\left(\frac{1-\epsilon}{1-\theta(\mathit{ext}(h))} + \epsilon\right) + |\mathrm{FP}_h| \lg(\epsilon) + |\mathrm{FN}_h| \lg(\epsilon) \end{aligned}$$

D may be broken down into three parts. First, a constant part which is due to the priors. Second, a part which is due to a sum of p independent random variables which arises from the p positive examples. These are labelled X_i $(i = 1, 2, \ldots, p)$ where X_i is the random variable corresponding to the additive component due to the ith positive example. Third, a part which is due to a sum of n independent random variables which arises from the n negative examples, which are similarly labelled with Y_i $(i = 1, 2, \ldots, n)$.

$$
\begin{aligned}
D = & \lg(P(h_t)) - \lg(P(h_\delta)) \\
& + |\mathrm{TP}_{h_t}| \lg(\frac{1-\epsilon}{\theta(ext(h_t))} + \epsilon) - |\mathrm{TP}_{h_\delta}| \lg(\frac{1-\epsilon}{\theta(ext(h_\delta))} + \epsilon) \\
& + |\mathrm{FP}_{h_t}| \lg(\epsilon) - |\mathrm{FP}_{h_\delta}| \lg(\epsilon) \\
& + |\mathrm{TN}_{h_t}| \lg(\frac{1-\epsilon}{1-\theta(ext(h_t))} + \epsilon) - |\mathrm{TN}_{h_\delta}| \lg(\frac{1-\epsilon}{1-\theta(ext(h_\delta))} + \epsilon) + \\
& + |\mathrm{FN}_{h_t}| \lg(\epsilon) - |\mathrm{FN}_{h_\delta}| \lg(\epsilon) \\
= & \lg(P(h_t)) - \lg(P(h_\delta)) \\
& + X_1 + X_2 + \cdots + X_p \\
& + Y_1 + Y_2 + \cdots + Y_n
\end{aligned}
\tag{7.1}
$$

To simplify the notation let $\theta_t = \theta(ext(h_t))$. Also, $\theta(ext(h_\delta))$ may be expressed in terms of θ_t, δ_a, and δ_b, so $\theta(ext(h_\delta)) = \theta_t - \delta_a + \delta_b$. The way positive examples are given to the learner places a probability measure over the examples, which is dependent on the target concept c. Consider a positive example e_i given to the learner. The probability $P(e_i)$ may be calculated as:

$$
P(e_i) = \begin{cases} D_X(e_i)(\frac{1-\mu_p}{\theta(c)} + \mu_p) & \text{if } e_i \in c, \\ D_X(e_i)(\mu_p) & \text{otherwise.} \end{cases}
$$

X_i is a random variable, that maps an example to the value the example contributes to the sum D. The value e_i maps to, through the random variable X_i, depends on its membership of $ext(h_t) = c$ and $ext(h_\delta)$. This mapping is shown in the following table:

Condition on e_i	Value given to e_i by random variable X_i
$e_i \in ext(h_t) \wedge e_i \in ext(h_\delta)$	$\lg(\frac{1-\epsilon}{\theta_t} + \epsilon) - \lg(\frac{1-\epsilon}{\theta_t - \delta_a + \delta_b} + \epsilon)$
$e_i \in ext(h_t) \wedge e_i \notin ext(h_\delta)$	$\lg(\frac{1-\epsilon}{\theta_t} + \epsilon) - \lg(\epsilon)$
$e_i \notin ext(h_t) \wedge e_i \in ext(h_\delta)$	$\lg(\epsilon) - \lg(\frac{1-\epsilon}{\theta_t - \delta_a + \delta_b} + \epsilon)$
$e_i \notin ext(h_t) \wedge e_i \notin ext(h_\delta)$	$\lg(\epsilon) - \lg(\epsilon) = 0$

Note, the range R_X of this random variable equals $\{\lg(\frac{1-\epsilon}{\theta_t} + \epsilon) - \lg(\frac{1-\epsilon}{\theta_t - \delta_a + \delta_b} + \epsilon), \lg(\frac{1-\epsilon}{\theta_t} + \epsilon) - \lg(\epsilon), \lg(\epsilon) - \lg(\frac{1-\epsilon}{\theta_t - \delta_a + \delta_b} + \epsilon), 0\}$. The probability that X_i takes one of these values may now be calculated.

$$
\begin{aligned}
P(X_i = \lg(\tfrac{1-\epsilon}{\theta_t} + \epsilon) - \lg(\tfrac{1-\epsilon}{\theta_t - \delta_a + \delta_b} + \epsilon)) &= \mu_p(\theta_t - \delta_a) + (1-\mu_p)(1 - \tfrac{\delta_a}{\theta_t}) \\
P(X_i = \lg(\tfrac{1-\epsilon}{\theta_t} + \epsilon) - \lg(\epsilon)) &= \mu_p \delta_a + (1-\mu_p)\tfrac{\delta_a}{\theta_t} \\
P(X_i = \lg(\epsilon) - \lg(\tfrac{1-\epsilon}{\theta_t - \delta_a + \delta_b} + \epsilon)) &= \mu_p \delta_b \\
P(X_i = 0) &= \mu_p(1 - (\theta_t + \delta_b))
\end{aligned}
$$

The expected value of X_i is now calculated and shown to be greater than zero for the following conditions:

- $0 \leq \epsilon < 1$,
- $0 \leq \mu_p \leq \epsilon$,

- $0 < \theta_t \leq 1$,
- $0 < \delta_a + \delta_b$,
- $0 \leq \delta_a \leq \theta_t$, and
- $0 \leq \delta_b \leq 1 - \theta_t$.

Note, that $\theta_t = 0$ is not included as the process that generates the positive examples will not work if $\theta_t = 0$. Similarly, $\theta_t = 1$ is not considered if addressing the negative examples. Let $e_p(\epsilon, \mu_p, \theta_t, \delta_a, \delta_b) = E(X_i) = \sum_{v \in R_X} vP(X_i = v)$.

$$\begin{aligned}
e_p(\epsilon, \mu_p, \theta_t, \delta_a, \delta_b) =& (1 - \mu_p + \mu_p\theta_t)\lg(\frac{1-\epsilon}{\theta_t} + \epsilon) \\
&- ((1-\mu_p)(1 - \frac{\delta_a}{\theta_t}) + \mu_p(\theta_t - \delta_a) + \mu_p\delta_b))\lg(\frac{1-\epsilon}{\theta_t - \delta_a + \delta_b} + \epsilon) \\
&- ((1-\mu_p)\frac{\delta_a}{\theta_t} + \mu_p(\delta_a - \delta_b))\lg(\epsilon)
\end{aligned}$$

By Lemma B.0.1, e_p is greater than zero for the above conditions. Hence, for any comparison between h_t and h_δ there exists a $\beta_p > 0$ where $\beta_p = E(X_i) = e_p(\epsilon, \mu_p, \theta_t, \delta_a, \delta_b)$. By symmetry there also exists a $\beta_n > 0$ where $\beta_n = E(Y_i)$.

As R_X, the range of the random variable X_i, is a set of four real finite values R_X may be bound between two constants. Let $a_x = \min(R_X)$ and let $b_x = \max(R_X)$. Hence, the range of X_i is bound: $a_x \leq X_i \leq b_x \quad (i = 1, 2, \cdots, p)$. Similarly the range of Y_i is also bound: $a_y \leq Y_i \leq b_y \quad (i = 1, 2, \cdots, n)$. Let $Z_i = X_i \quad (i = 1, 2, \cdots, p)$, $Z_{i+p} = Y_i \quad (i = 1, 2, \cdots, n)$, so $Z_1, Z_2, \cdots, Z_m$ are independent and $a_x \leq Z_i \leq b_x \quad (i = 1, 2, \ldots, p)$ and $a_y \leq Z_i \leq b_y \quad (i = p+1, p+2, \ldots, m)$. The expected value of $\overline{Z} = \frac{Z_1 + Z_2 + \cdots + Z_m}{m}$ may be calculated as $E(\overline{Z}) = \frac{p\beta_p + n\beta_n}{m}$.

Let $t = \frac{p\beta_p + n\beta_n}{2m}$, then by Hoeffding's inequality we have:

$$\begin{aligned}
\Pr\{\overline{Z} - E(\overline{Z}) \leq -\frac{p\beta_p + n\beta_n}{2m}\} &\leq e^{\left(\frac{-2m^2(\frac{p\beta_p + n\beta_n}{2m})^2}{p(b_x - a_x)^2 + n(b_y - a_y)^2}\right)} \\
&= e^{\left(\frac{-(p\beta_p + n\beta_n)^2}{2(p(b_x - a_x)^2 + n(b_y - a_y)^2)}\right)} \\
&\leq e^{-\frac{m\beta^2}{2d^2}} \\
&= r^m
\end{aligned} \tag{7.2}$$

where $\beta = \min\{\beta_p, \beta_n\}$, $d = \max\{(b_x - a_x), (b_y - a_y)\}$, and $r = e^{-\frac{\beta^2}{2d^2}} < 1$. Note that, geometric series, $\sum_m r^m$, converges.

Recall that $D = \lg(P(h_t)) - \lg(P(h_\delta)) + \sum_{i=1}^{m} Z_i$. Now, if $\overline{Z} > \frac{p\beta_p + n\beta_n}{2m}$ then $D > 0$ when $m > m_{i_\delta}$, where $m_{i_\delta} = \lceil \frac{-2(\lg(P(h_t)) - \lg(P(h_\delta)))}{\beta} \rceil$. Hence using Equation 7.2, an upper bound on the probability that the incorrect hypothesis is selected will be:

$$Pr\{\text{the incorrect hypthesis is selected in round } m\} \leq \begin{cases} 1 & \text{if } m \leq m_{i_\delta}, \\ r^m & \text{otherwise.} \end{cases}$$

Clearly the sum $\sum_{m \in \mathbb{N}} Pr\{\text{the incorrect hypthesis is selected in round } m\}$ is finite.

The proof is now extended to a finite class of concepts C, and the Borel-Cantelli lemma is applied to show the final result.

Each concept $c_i \in C$ has a set of hypotheses $H_{c_i} = \{h \in H|\text{error}(ext(h), c_i) = 0\}$ which represent it. Given any set of training examples the Q heuristic will produce the same ordering on H_{c_i} where the hypothesis (or set of hypotheses) with the highest Q value will be the hypothesis (or set of hypotheses) with the highest prior probability. Indeed, the prior component is the only part that varies for different hypotheses that represent the same concept. From among the hypotheses with maximum prior probability there is one with the lowest syntactic order. Let this hypothesis be denoted h_i. Hence, if Algorithm 12 selects a hypothesis that represents the concept c_i it will always select h_i over any other hypothesis in H_{c_i}. As the concept class C is finite we only need to compare the Q value over a finite set of hypotheses, giving the maximum Q value over the possible infinite hypothesis space H.

Suppose c_t is the target concept. Then there is a hypothesis h_t which represents c_t and is first in the syntactic ordering of the maximal prior hypotheses. Consider a comparison of h_t and h_i where $t \neq i$. Let the probability that $Q(h_t) \leq Q(h_i)$ after m examples be $p_{i,m}$. Then the probability after m training examples that there exists an $i \in \{1, \ldots, k\} - \{t\}$ such that $Q(h_t) \leq Q(h_i)$ will be at most $\sum_{i \in \{1,\ldots,k\}-\{t\}} p_{i,m}$. As $\sum_{m=0}^{\infty} p_{i,m}$ is finite for each $i \in \{1, \ldots, k\} - \{t\}$ the sum $\sum_{m=0}^{\infty}(\sum_{i \in \{1,\ldots,k\}-\{t\}} p_{i,m})$ will also be finite. With probability 1, by the Borel-Cantelli lemma, h_t will have the highest Q all but finitely many times. As the algorithm selects the hypothesis with the highest Q value (and smallest index if there are more than one) it will with probability 1 induce the correct hypothesis all but finitely many times. Hence, Algorithm 12 stochastically identifies C in the limit. ■

7.2.2 A restricted infinite concept class

The problem with extending the above result to an infinite concept class is when the hypotheses, with respect to their priors, converge on the target concept too quickly. When this occurs over an infinite set of concepts the bound on inducing an incorrect hypothesis is not finite. To address this problem a restriction is placed on the rate any hypothesis may be converged on.

Definition 7.2.1 (g-sparse with respect to a concept) *Let $g : \mathbb{N} \to \mathbb{R}$. Let c be a concept. A hypothesis space, $H = \{h_i | i \in \mathbb{N}\}$, is said to be* g-sparse *with respect to concept c if there exists $m_c \in \mathbb{N}$ and $c_c \in \mathbb{R}$ such that for all $j > m_c$, we have:*

$$error(c, ext(h_j)) \neq 0 \Rightarrow error(c, ext(h_j)) \geq c_c g(j).$$

Definition 7.2.2 (g-sparse) *A hypothesis space H is said to be* g-sparse *if H is* g-sparse *with respect to concepts $\emptyset$, the instance space X, and $ext(h_i)$ for all $h_i \in H$.*

Theorem 7.2.2 *Let C be any concept class and H be any hypothesis space which is complete for C. Let $\epsilon \in [0, 1)$ be the noise parameter. Assuming H is* g-sparse *where $g(i) = \frac{1}{i^\alpha}$ for $\alpha < 1$, there exists an algorithm that stochastically identifies C in the limit with respect to H when examples are provided by $GEN^O_{\langle \mu_p, \mu_n \rangle}$ for any oracle O and any $\mu_p \leq \epsilon$ and $\mu_n \leq \epsilon$.*

PROOF:

This proof extends the previous proof. Similarly, the learner used is the one described in Algorithm 12. The priors on the hypothesis space are set to be the geometric sequence with respect to an enumeration of the hypothesis space, that is $P(h_i) = (\frac{1}{2})^i$. We could have also used any prior where for all but finitely many $i \in \mathbb{N}$ and $p \in (0,1)$ we have:

$$P(h_i) \leq (\frac{1-p}{p})p^i$$

However, for simplicity $P(h_i) = (\frac{1}{2})^i$ is used. Note the priors on the hypotheses are effectively chosen by a the learner. Although it would be desirable that the priors reflect the probability a concept is selected to be the target concept, this is not essential as the learner must induce a correct hypothesis for any concept. That is, the criteria of success for the stochastic limit result requires the learner to correctly converge on a target hypothesis for any concept, hence, there is no assumption made regarding the distribution of concept selection. Correspondingly the prior probability, which may be decided by and is part of the learning algorithm, must work on each concept individually. One aim for future research is to consider complexity issues, in which case both the distribution of concepts presented to the learner and the prior probability used become critical.

Once again we consider the difference between the Q-value of a correct hypothesis h_t and an incorrect hypothesis h_δ. Let $D = Q(h_t) - Q(h_\delta)$, recall Equation 7.1.

$$\begin{aligned} D = & \lg(P(h_t)) - \lg(P(h_\delta)) \\ & + X_1 + X_2 + \cdots + X_p \\ & + Y_1 + Y_2 + \cdots + Y_n \end{aligned}$$

The expected value of the random variable X_i may be calculated as $E(X_i) = \beta_p$ which is positive as shown by Lemma B.0.1. Similarly for Y_i, $E(Y_i) = \beta_n > 0$.

We may now apply Hoeffding's inequality to find upper bounds on the probability of inducing the incorrect hypothesis. These bounds are split into two possible cases: Case 1, where $P(h_t) \leq P(h_\delta)$; and Case 2, where $P(h_t) > P(h_\delta)$. Recall that the prior probability over the hypothesis space is assumed to be a geometric series: $P(h_i) = (\frac{1}{2})^i$. Let i_t be the index of h_t and i_δ be the index of h_δ. This means Case 1 corresponds to $i_\delta \in \{1, 2, \ldots, i_t\}$ and Case 2 to $i_\delta \in \{i_t + 1, i_t + 2, \ldots\}$.

Case 1: $P(h_t) \leq P(h_\delta)$

Let $t = \frac{p\beta_p + n\beta_n}{2m}$. Recall from Equation 7.2 that by Hoeffding's inequality we have:

$$\Pr\{\overline{Z} - E(\overline{Z}) \leq -\frac{p\beta_p + n\beta_n}{2m}\} \leq r^m$$

where $\beta = \min\{\beta_p, \beta_n\}$, $d = \max\{(b_x - a_x), (b_y - a_y)\}$, and $r = e^{-\frac{\beta^2}{2d^2}} < 1$. Note, this geometric series converges.

If $\overline{Z} > \frac{p\beta_p + n\beta_n}{2m}$ then $D > 0$ when $m > m_{i_\delta}$, where $m_{i_\delta} = \lceil\frac{-2(\lg(P(h_t)) - \lg(P(h_\delta)))}{\beta}\rceil$. Hence, an upper bound on the probability that the incorrect hypothesis is selected will be:

$$\text{bound}_1(i,m) = \begin{cases} 1 & \text{if } m \leq m_{i_\delta}, \\ r^m & \text{otherwise.} \end{cases}$$

So, $Pr\{D \leq 0\} \leq \text{bound}_1(i,m)$. The sum $\sum_{i=1}^{i_t} \sum_{m=1}^{\infty} \text{bound}_1(i,m)$ clearly sums over to a finite value.

Case 2: $P(h_t) > P(h_\delta)$

In this case the larger prior on h_t 'helps' the learner select the correct hypothesis. This allows the bound on the probability of selecting the incorrect hypothesis to be smaller.

The hypothesis h_δ is incorrectly selected over h_t when $Q(h_t) < Q(h_\delta)$ that is when $D < 0$. This implies that $D \leq 0$. Let $B = \lg(P(h_t)) - \lg(P(h_\delta)) > 0$.

$$\begin{aligned} D &\leq 0 \\ Z_1 + Z_2 + \cdots + Z_m + B &\leq 0 \\ \frac{Z_1 + Z_2 + \cdots + Z_m}{m} &\leq -\frac{B}{m} \\ \overline{Z} - E(\overline{Z}) &\leq -E(\overline{Z}) - \frac{B}{m} \\ \overline{Z} - E(\overline{Z}) &\leq -(\frac{p\beta_p + n\beta_n}{m}) - \frac{B}{m} \end{aligned}$$

Now, by applying the Hoeffding's inequality, an upper bound on the probability that h_δ is selected over h_t may be calculated.

$$\begin{aligned} \Pr\{\overline{Z} - E(\overline{Z}) \leq -(\frac{p\beta_p + n\beta_n}{m}) - \frac{B}{m}\} &\leq e^{(\frac{-2m^2(-(\frac{p\beta_p + n\beta_n}{m}) - \frac{B}{m})^2}{p(b_x - a_x)^2 + n(b_y - a_y)^2})} \\ &= e^{(\frac{-2(-(p\beta_p + n\beta_n) - B)^2}{p(b_x - a_x)^2 + n(b_y - a_y)^2})} \\ &\leq e^{(\frac{-2(-(m\beta) - B)^2}{md^2})} \\ &= e^{(\frac{-2(m^2\beta^2 + 2m\beta B + B^2)}{md^2})} \\ &= e^{(\frac{-2(m\beta^2 + 2\beta B + \frac{B^2}{m})}{d^2})} \end{aligned}$$

where similarly $\beta = \min\{\beta_p, \beta_n\}$, $d = \max\{(b_x - a_x), (b_y - a_y)\}$. We may now simplify B.

$$\begin{aligned} B &= \lg(P(h_t)) - \lg(P(h_\delta)) \\ &= \lg((\frac{1}{2})^{i_t}) - \lg((\frac{1}{2})^{i_\delta}) \\ &= i_\delta - i_t \end{aligned}$$

Note, β and d are functions of the index i of the hypothesis h_δ and hence β will be rewritten as $\beta(i)$ and d will be rewritten as $d(i)$. This is important when we consider the infinite concept class. Let $P(\text{Bad}(m,i))$ denote the probability h_δ is selected over h_t in round m. Note, h_δ is indexed by i. This is the probability that in round m the wrong hypothesis is selected out of h_δ and h_t.

Now, this probability may be bound as follows:

$$P(\text{Bad}(m,i)) \le e^{\left(\frac{-2(m\beta(i)^2+2\beta(i)(i-i_t)+\frac{(i-i_t)^2}{m})}{d(i)^2}\right)} \tag{7.3}$$

Let $\text{bound}_2(i,m) = e^{\left(\frac{-2(m\beta(i)^2+2\beta(i)(i-i_t)+\frac{(i-i_t)^2}{m})}{d(i)^2}\right)}$.

This completes the section of the proof that provides a bound on the probability of the Q heuristic selecting the incorrect hypothesis h_δ from either h_t or h_δ. Note that, when $\text{error}(h_\delta, h_t) = 0$ a stricter bound than that of Equation 7.3 may be obtained, namely $P(\text{Bad}(m,i)) \le 0$ as h_t will always be selected over h_δ in this case. Let $\text{bound}_3(i,m) = 0$.

We must sum $P(\text{Bad}(m,i))$, showing it to be finite, and apply the Borel-Cantelli lemma to show that with probability 1 the algorithm only induces an incorrect hypothesis a finite number of times. Hence, we are required to prove that the sum below is finite.

$$\sum_{i=1}^{\infty}\sum_{m=1}^{\infty} P(\text{Bad}(i,m)) = \sum_{i=1}^{i_t}\sum_{m=1}^{\infty} P(\text{Bad}(i,m)) + \sum_{i=i_t+1}^{\infty}\sum_{m=1}^{\infty} P(\text{Bad}(i,m))$$

Clearly $\sum_{i=1}^{i_t}\sum_{m=1}^{\infty} P(\text{Bad}(i,m))$ is finite as $\sum_{m=1}^{\infty}\text{bound}_1(i,m)$ is finite for any $i \in \{1, 2, \ldots, i_t\}$. Consider the second sum:

$$\sum_{i=i_t+1}^{\infty}\sum_{m=1}^{\infty} P(\text{Bad}(i,m))$$

As the hypothesis space is g-sparse for the target hypothesis h_t there exists constants $m_t > 4i_t$ and c_t such that for all $j > m_t$

$$\text{error}(ext(h_i), ext(h_j)) \neq 0 \Rightarrow \text{error}(ext(h_i), ext(h_j)) \ge c_t g(j)$$

Also, from Lemma B.0.2 there exists constants $c_{p1} > 0$ and $c_{p2} > 0$ such that for all $\delta \le c_{p1}$, $e_p(\epsilon, \mu_p, \theta_t, \delta_a, \delta - \delta_a) \ge c_{p2}\delta$, where $0 \le \delta_a \le \delta$. By symmetry there also exists constants $c_{n1} > 0$ and $c_{n2} > 0$ such that for all $\delta \le c_{n1}$, $e_n(\epsilon, \mu_p, \theta_t, \delta_a, \delta - \delta_a) \ge c_{n2}\delta$, where $0 \le \delta_a \le \delta$. Let $c_1 = \min\{c_{p1}, c_{n1}\}$ and $c_2 = \min\{c_{p2}, c_{n2}\}$. Clearly, for all $\delta \le c_1$, $e_p(\epsilon, \mu_p, \theta_t, \delta_a, \delta - \delta_a) \ge c_2\delta$, where $0 \le \delta_a \le \delta$. And for all $\delta \le c_1$, $e_n(\epsilon, \mu_p, \theta_t, \delta_a, \delta - \delta_a) \ge c_2\delta$, where $0 \le \delta_a \le \delta$. As $\beta(i) = \min\{e_p(\epsilon, \mu_p, \theta_t, \delta_a, \delta - \delta_a), e_n(\epsilon, \mu_p, \theta_t, \delta_a, \delta - \delta_a)\}$ and $\delta = \text{error}(ext(h_i), ext(h_j))$ then $\beta(i) \ge c_2(\text{error}(ext(h_t), ext(h_i)))$.

Also note, by Lemma B.0.7, for all $\delta > c_1$, $e_p(\epsilon, \mu_p, \theta_t, \delta_a, \delta - \delta_a) \ge c_2c_1$, where $0 \le \delta_a \le \delta$. Correspondingly, for all $\delta > c_1$, $e_n(\epsilon, \mu_p, \theta_t, \delta_a, \delta - \delta_a) \ge c_2c_1$, where $0 \le \delta_a \le \delta$.

The sum is now partitioned into four separate sums.

$$\begin{aligned}\sum_{i=i_t+1}^{\infty}\sum_{m=1}^{\infty}P(\mathrm{Bad}(i,m)) =& \sum_{i\in S_1}\sum_{m=1}^{\infty}P(\mathrm{Bad}(i,m)) +\\ &\sum_{i\in S_2}\sum_{m=1}^{\infty}P(\mathrm{Bad}(i,m)) +\\ &\sum_{i\in S_3}\sum_{m=1}^{\infty}P(\mathrm{Bad}(i,m)) +\\ &\sum_{i\in S_4}\sum_{m=1}^{\infty}P(\mathrm{Bad}(i,m))\\ =&\mathrm{Sum}_1+\mathrm{Sum}_2+\mathrm{Sum}_3+\mathrm{Sum}_4\end{aligned}$$

where

- $S_1 = \{i_t+1, i_t+2, \ldots, m_t\}$,
- $S_2 = \{j \in \{m_t+1, \ldots\} \mid \mathrm{error}(\mathit{ext}(h_t), \mathit{ext}(h_j)) = 0\}$,
- $S_3 = \{j \in \{m_t+1, \ldots\} \mid 0 < \mathrm{error}(\mathit{ext}(h_t), \mathit{ext}(h_j)) \leq c_1\}$, and
- $S_4 = \{j \in \{m_t+1, \ldots\} \mid c_1 < \mathrm{error}(\mathit{ext}(h_t), \mathit{ext}(h_j))\}$.

Each of these sums is shown to be finite.

$\mathbf{Sum}_1$ = $\sum_{i\in S_1}\sum_{m=1}^{\infty}P(\mathrm{Bad}(i,m))$: As S_1 is finite and $\sum_{m=1}^{\infty}\mathrm{bound}_2(i,m)$ is finite for any i, the entire sum will clearly be finite.

$\mathbf{Sum}_2$ = $\sum_{i\in S_2}\sum_{m=1}^{\infty}P(\mathrm{Bad}(i,m))$: As the target hypothesis h_t and the "incorrect" hypothesis h_δ are identical with respect to the instance space distribution, the nonprior part of the Q-heuristic will be identical and hence h_t will always be chosen as it has the higher prior probability. We have:

$$P(\mathrm{Bad}(i,m)) \leq \mathrm{bound}_3(i,m).$$

Hence, the probability of inducing the incorrect hypothesis is zero. Moreover, this sum will be finite.

$\mathbf{Sum}_3$ = $\sum_{i\in S_3}\sum_{m=1}^{\infty}P(\mathrm{Bad}(i,m))$: This sum is considering all the hypotheses whose *error* is nonzero and less than or equal to c_1. We know that $\mathrm{error}(\mathit{ext}(h_t), \mathit{ext}(h_i)) \geq c_t g(i)$ where $g(i) = \frac{1}{i^\alpha}$ for some $\alpha < 1$. And $\beta(i) \geq c_2(\mathrm{error}(\mathit{ext}(h_t), \mathit{ext}(h_i)))$. Hence, $\beta(i) \geq \frac{c_2 c_t}{i^\alpha}$. The bound as previously calculated is:

$$\mathrm{bound}_2(i,m) = e^{\left(\frac{-2(m\beta(i)^2+2\beta(i)(i-i_t)+\frac{(i-i_t)^2}{m})}{d(i)^2}\right)} \tag{7.4}$$

Let $d_b = \max\{\lg(\frac{1-\epsilon}{\theta_t}+\epsilon) - 2\lg(\epsilon) + \lg(\frac{1-\epsilon}{\theta_t - c_1}+\epsilon), \lg(\frac{1-\epsilon}{1-\theta_t}+\epsilon) - 2\lg(\epsilon) + \lg(\frac{1-\epsilon}{1-\theta_t-c_1}+\epsilon)\}$. As error($ext(h_t), ext(h_i)) \leq c_1$, $d(i) \leq d_b$. Hence:

$$
\begin{aligned}
\text{bound}_2(i,m) &\leq e^{\left(\frac{-2(m\beta(i)^2 + 2\beta(i)(i-i_t) + \frac{(i-i_t)^2}{m})}{d_b^2}\right)} \\
&\leq e^{\left(\frac{-2(\frac{m c_2^2 c_t^2}{i^{2\alpha}} + 2\frac{c_2 c_t (i-i_t)}{i^\alpha} + \frac{(i-i_t)^2}{m})}{d_b^2}\right)} \\
&\leq e^{\left(\frac{(-2\frac{m c_2^2 c_t^2}{i^{2\alpha}} - 4 c_2 c_t i^{-\alpha}(i-i_t) - 2\frac{(i-i_t)^2}{m})}{d_b^2}\right)} \\
&\leq e^{\left(\frac{(-2\frac{m c_2^2 c_t^2}{i^{2\alpha}} - c_2 c_t i^{1-\alpha} - \frac{i^2}{m})}{d_b^2}\right)} \qquad \text{(for } i > 4i_t\text{)} \\
&\leq e^{-c_3(\frac{m}{i^{2\alpha}} + i^{1-\alpha} + \frac{i^2}{m})} \qquad \text{(where } c_3 = \min\{2c_2^2 c_t^2, c_2 c_t, 1\}/d_b^2\text{)} \\
&\leq e^{-c_3(i^{1-\alpha} + m^{\frac{1-\alpha}{1+\alpha}})} \qquad \text{(Lemma B.0.8)}
\end{aligned}
$$

$$
\begin{aligned}
\sum_{i \in S_3} \sum_{m=1}^{\infty} \text{bound}_2(i,m) &\leq \sum_{i \in S_3} \sum_{m=1}^{\infty} e^{-c_3(i^{1-\alpha} + m^{\frac{1-\alpha}{1+\alpha}})} \\
&= \sum_{i \in S_3} \sum_{m=1}^{\infty} e^{-c_3 i^{1-\alpha}} e^{-c_3 m^{\frac{1-\alpha}{1+\alpha}}} \\
&\leq \sum_{i=1}^{\infty} \sum_{m=1}^{\infty} e^{-c_3 i^{1-\alpha}} e^{-c_3 m^{\frac{1-\alpha}{1+\alpha}}} \\
&\leq \sum_{i=1}^{\infty} e^{-c_3 i^{1-\alpha}} \sum_{m=1}^{\infty} e^{-c_3 m^{\frac{1-\alpha}{1+\alpha}}}
\end{aligned}
$$

Hence the sum on the bound is finite (Lemma B.0.3).

$\text{Sum}_4 = \sum_{i \in S_4} \sum_{m=1}^{\infty} P(\text{Bad}(i,m))$: This sum considers the hypotheses whose *error* is greater than c_1. We know that $\beta(i) \geq c_2 c_1$ as error($ext(h_t), ext(h_i)) > c_1$. Recall $\text{bound}_2(i,m)$ in Equation 7.4 which used to bound elements in this sum.

We would expect it to be easier to show that Sum_4 converges than that of Sum_4 as the incorrect hypotheses in S_4 have larger *error* than that of S_3. However, as the range the random variable Z maps to is not bounded, we can not use d_b to bound $d(i)$. We note that $d(i)$ becomes large when either $\theta(h_i)$ is close to 0 or 1. Now as hypothesis space H is *g-sparse* with respect to $\emptyset$ and X, this places a bound on the rate $\theta(h_i)$ approaches either 0 or 1 consequently placing a bound on $d(i)$ as i increases.

$$
\begin{aligned}
d(i) &= \max\{\lg(\frac{1-\epsilon}{\theta_t}+\epsilon) - 2\lg(\epsilon) + \lg(\frac{1-\epsilon}{\theta(h_i)}+\epsilon), \\
&\qquad \lg(\frac{1-\epsilon}{1-\theta_t}+\epsilon) - 2\lg(\epsilon) + \lg(\frac{1-\epsilon}{1-\theta(h_i)}+\epsilon)\} \\
&= c_4 + \max\{\lg(\frac{1-\epsilon}{\theta(h_i)}+\epsilon), \lg(\frac{1-\epsilon}{1-\theta(h_i)}+\epsilon)\} \\
&\qquad (\text{where } c_4 = \lg(\frac{1-\epsilon}{\theta_t}+\epsilon) - 2\lg(\epsilon)) \\
&\leq c_4 + \max\{\lg(\frac{1-\epsilon}{(\frac{c_5}{i^\alpha})}+\epsilon), \lg(\frac{1-\epsilon}{(\frac{c_6}{i^\alpha})}+\epsilon)\} \\
&\qquad (\text{for } i > m_5 \text{ where } c_5 > 0,\ c_6 > 0, \text{ and } \alpha \in (0,1)) \\
&\leq c_4 + \max\{\lg(c_7 i^\alpha), \lg(c_8 i^\alpha)\} \quad (\text{for } i > m_6 \text{ where } c_7 > 0,\ c_8 > 0) \\
&= c_4 + \max\{\lg(c_7), \lg(c_8)\} + \alpha\lg(i) \\
&\leq \lg(i) \quad (\text{for } i > m_7)
\end{aligned}
$$

We can now consider the bound in question:

$$
\begin{aligned}
\text{bound}_2(i,m) &\leq e^{(\frac{-2(m\beta(i)^2 + 2\beta(i)(i-i_t) + \frac{(i-i_t)^2}{m})}{d(i)^2})} \\
&\leq e^{(\frac{-2(m c_2^2 c_1^2 + 2c_2 c_1 (i-i_t) + \frac{(i-i_t)^2}{m})}{(\lg(i))^2})} \\
&\leq e^{(\frac{-2(m c_2^2 c_1^2 + c_2 c_1 i)}{(\lg(i))^2})} \qquad (\text{For } i > i_t) \\
&\leq e^{(\frac{-2(m c_2^2 c_1^2 + c_2 c_1 i)}{(\lg(i))^2})} \qquad (\text{For } i > i_t) \\
&\leq e^{(\frac{-c_9(m+i)}{(\lg(i))^2})} \\
&\leq e^{-c_{10}(m^{\frac{1}{4}} + i^{\frac{1}{4}})} \qquad (\text{Lemma B.0.9})
\end{aligned}
$$

The sum $\sum_i^\infty \sum_{m=1}^\infty e^{-c_{10}(m^{\frac{1}{4}}+i^{\frac{1}{4}})}$ may be shown to be finite in the same way the previous sum is shown to be finite.

Hence, the entire sum is finite as all four sums are finite. Moreover, the Q-heuristic induces an incorrect hypothesis at most a finite number of times with probability 1. ■

7.3 Example Concept Classes

The learnability results presented in the previous two sections are interesting because our model incorporates noise in the data and a stochastic criterion of success. We feel that our approach is more realistic because although the classes discussed previously are learnable in the limit (in the traditional Gold [42] sense), they are not learnable in the Gold setting if noise is present. We next consider a class that is not learnable in the limit from positive only data in Gold's setting, but is learnable in our stochastic setting from only positive data even in the presence of noise.

Theorem 7.3.1 *Let* $H = \{h_1, h_2, h_3, \dots\} = \{\mathbb{N}, \emptyset, \{1\}, \{2\}, \{1,2\}, \{3\}, \{1,3\}, \dots\}$*. The concept* *class consisting of all the finite subsets of* $\mathbb{N}$ *together with* $\mathbb{N}$ *is stochastically learnable in the limit with respect to* H.[5]

PROOF: Let the instance space X be $\mathbb{N}$. Let C be the concept class defined in the statement of the theorem. We have $ext(h_1) = \mathbb{N}$ and for $i > 1$, $ext(h_i) = \text{bits}(i-2)$, where $\text{bits}(w)$ is a finite set of natural numbers where membership is defined as follows:

$$x \in \text{bits}(w) \Leftrightarrow (w \;\text{ div }\; 2^{x-1}) \mod 2 = 1$$

Let the instance space distribution $D_X(x) = \frac{6}{\pi^2 x^2}$.

We now show this hypothesis space to be *g-sparse*, where $g(x) = \frac{1}{(\ln(x))^2}$. This requires us to show that for any $h_i \in H$ there exists constants m_i and c_i such that for all h_j where $j > m_i$ we have:

$$\text{error}(ext(h_i), ext(h_j)) \neq 0 \Rightarrow \text{error}(ext(h_i), ext(h_j)) \geq c_i g(j)$$

First, consider h_1, so $ext(h_1) = \mathbb{N}$. Let $m_1 = 2$ and $c_1 = \frac{3(\ln(2))^2}{2\pi^2}$. Note, $ext(h_{2^k+1}) = \{1, 2, \ldots, k\}$. Now, for some $k \in \mathbb{N}$ and $2^k + 1 \leq j < 2^{k+1} + 1$ the *error* between h_1 and h_j will be least when $j = 2^k + 1$. A bound on this *error* may be calculated as $k + 1 \notin ext(h_{2^k+1})$.

$$\begin{aligned}
\text{error}(ext(h_1), ext(h_{2^k+1})) &\geq \frac{6}{\pi^2(k+1)^2} \\
&\geq \frac{6}{\pi^2(2k)^2}, \qquad \text{for } k \geq 1 \\
&= \frac{3(\ln(2))^2}{2\pi^2} \frac{1}{k^2(\ln(2))^2} \\
&\geq \frac{3(\ln(2))^2}{2\pi^2} \frac{1}{(\ln(2^k+1))^2} \\
&= c_1 g(2^k+1)
\end{aligned}$$

As the *error* is least over the range $2^k + 1 \leq j < 2^{k+1} + 1$ when $j = 2^k + 1$, clearly $\text{error}(ext(h_1), ext(h_j)) \geq c_1 g(2^k+1)$ for $2^k+1 \leq j < 2^{k+1}+1$. Moreover, $\text{error}(ext(h_1), ext(h_j)) \geq c_1 g(j)$ for $2^k+1 \leq j < 2^{k+1}+1$ as g is a strictly decreasing function. Hence, $\text{error}(ext(h_1), ext(h_j)) \geq c_1 g(j)$ for all $j > m_1$ as $2^k + 1 \leq j < 2^{k+1} + 1$ partitions the set $\{3, 4, 5, \ldots\}$.

Second, consider h_i where $i > 1$. Then for some $k > \lceil \lg(i) \rceil$ and $i + 2^k \leq j < i + 2^{k+1}$ the least *error* between h_i and h_j will be reached when $j = i + 2^k$. Let $m_i = i + 2^{\lceil \lg(i) \rceil}$ and $c_i = \frac{6(\ln(2))^2}{\pi^2}$. Note, $ext(h_{i+2^k}) = ext(h_i) \cup \{k\}$. A bound on the *error* may be calculated as $k \in ext(h_{i+2^k})$ but

[5] *Since the data presentation could be guided by any oracle, this result holds for positive only data, too.*

$k \notin ext(h_i)$.

$$\begin{aligned}
\text{error}(ext(h_i), ext(h_{i+2^k})) &\geq \frac{6}{\pi^2 k^2} \\
&= \frac{6(\ln(2))^2}{\pi^2} \frac{1}{k^2(\ln(2))^2} \\
&= \frac{6(\ln(2))^2}{\pi^2} \frac{1}{(\ln(2^k))^2} \\
&\geq \frac{6(\ln(2))^2}{\pi^2} \frac{1}{(\ln(i+2^k))^2} \\
&= c_i g(i+2^k)
\end{aligned}$$

Similarly, $\text{error}(ext(h_1), ext(h_j)) \geq c_i g(j)$ for all $j > m_i$, as $i + 2^k \leq j < i + 2^{k+1}$ partitions the cases of interest and g is strictly decreasing.

As H is *g-sparse* where $g(i) = \frac{1}{(\ln(i))^2}$, H is also *g-sparse* where $g(i) = \frac{1}{i^{\frac{1}{2}}}$ given the former is a stronger bound. Finally, by Theorem 7.2.2 this concept class is stochastically learnable in the limit as it is *g-sparse*, where $g(i) = \frac{1}{i^\alpha}$ for $\alpha = \frac{1}{2}$. ■

We now consider the classes of concepts that consists of the empty set, the set of naturals and sets of the form $\{1, 2, \ldots, k\}$, this class is a subset of the class shown to be stochastically learnable in the limit in the previous theorem, thence, by Theorem 7.1.1, the class will also be stochastically learnable in the limit. However, we include this result as it may be proved using a restricted hypothesis space and a different instance space distribution which forms a tighter bound on the g-sparse restriction, and hence a more difficult concept to learn.

Theorem 7.3.2 *Let the instance space X be $\mathbb{N}$. Let the instance space distribution $D_X(x) = \frac{s_1}{x^{3/2}}$ where s_1 is the normalizing constant. Let $H = \{h_1, h_2, h_3, \ldots\} = \{\mathbb{N}, \emptyset, \{1\}, \{1,2\}, \{1,2,3\}, \{1,2,3,4\}, \ldots\}$. The concept class consisting of $\mathbb{N}$ and $\emptyset$ together with $\{\{1, 2, \ldots, k\} | k \in \mathbb{N}\}$ is stochastically learnable in the limit with respect to H.*

PROOF: Let C be the concept class defined in the statement of the theorem. Let the hypothesis space $H = \{h_1, h_2, h_3, \ldots\} = \{\mathbb{N}, \emptyset, \{1\}, \{1,2\}, \{1,2,3\}, \{1,2,3,4\}, \ldots\}$. So, $ext(h_1) = \mathbb{N}$ and for $i > 1$, $ext(h_i) = \{1, 2, \ldots, (i-2)\}$.

Now for $i > 1$ the concepts $ext(h_i)$ are clearly *g-sparse* as the *error* between h_i and h_j for $j > i$ increase as j increases. It remains to show that $ext(h_1)$ is *g-sparse*. By using integration we have the following bound:

$$\text{error}(ext(h_1), ext(h_j)) > \frac{2s_1}{(k-1)^{\frac{1}{2}}}.$$

By Theorem 7.2.2 this concept class is stochastically learnable in the limit. ■

We now consider a highly restricted first-order language. An example of a hypothesis in this language is given in Example 7.3.1.

Example 7.3.1 *An example of a logic program that represents the set consisting of the numbers*

2, 3, and numbers greater than or equal to 5 is as follows:

$$P(s(s(0))).$$
$$P(s(s(s(0)))).$$
$$P(s(s(s(s(s(X)))))).$$
$$P(s(s(s(s(s(s(s(X)))))))).$$

Note, the last clause is redundant in representing the concept.

Let $\mathcal{L}_{\text{nums}}$ be a first-order language consisting of atomic formulas with one unary predicate symbol P, one unary function symbol s, and one constant symbol 0. Let the instance space X_{nums} be equal to the Herbrand bases for $\mathcal{L}_{\text{nums}}$ We consider a hypothesis space that consists of all the logic programs in $\mathcal{L}_{\text{nums}}$ where the clauses are facts. In $\mathcal{L}_{\text{nums}}$ a clause can have two possible forms. First, it may contain the constant symbol 0 in which case it is of the form $P(s^k(0))$ for $k \geq 0$. Second, the clause may contain a variable X in which case it is of the form $P(s^k(X))$ for some $k \geq 0$. Note, $s^k(t)$ represents $\overbrace{s(s(\cdots s(}^{k}t)\cdots))$. Now we may enumerate the hypothesis space (note each hypothesis will be a set of clauses). Let $H_{\text{nums}} = \{h_1, h_2, h_3, \dots\}$ where

$$P(s^k(0)) \in h_i \Leftrightarrow (i-1 \;\; \text{div} \;\; 2^{2k}) \mod 2 = 1$$

and

$$P(s^k(X)) \in h_i \Leftrightarrow (i-1 \;\; \text{div} \;\; 2^{2k+1}) \mod 2 = 1$$

For example the logic program given in Example 7.3.1 is h_{34897} in this enumeration.

Theorem 7.3.3 *Given H_{nums}, X_{nums}. Let $D_{X_{nums}}(P(s^k(0))) = \frac{6}{\pi^2(k+1)^2}$. Then the class of concepts definable in H_{nums} are stochastically learnable in the limit with respect to H_{nums}.*

PROOF:

This hypothesis space is now shown to be *g-sparse*, where $g(x) = \frac{1}{(\ln(x))^2}$. So we are required to show that for any $h_i \in H_{\text{nums}}$ there exists constants m_i and c_i such that for all h_j where $j > m_i$ we have:

$$\text{error}(ext(h_i), ext(h_j)) \neq 0 \Rightarrow \text{error}(ext(h_i), ext(h_j)) \geq c_i g(j).$$

First, suppose that h_i does not contain clauses with variable symbols. Let $m_i = 2^{\lceil \lg(i-1) \rceil + 2}$ and $c_i = \frac{24(\ln(2))^2}{\pi^2}$. Then for $k \geq \frac{\lceil \lg(i-1) \rceil}{2} + 1$ and $i + 2^{2k} \leq j < i + 2^{2(k+1)}$ the least *error* between h_i and h_j will be reached when $j = i + 2^{2k}$. A bound on this error may now be calculated as follows:

$$\begin{aligned}
\text{error}(ext(h_i), ext(h_{i+2^{2k}})) &\geq \frac{6}{\pi^2 k^2} \\
&= \frac{24(\ln(2))^2}{\pi^2} \frac{1}{4(\ln(2))^2 k^2} \\
&= \frac{24(\ln(2))^2}{\pi^2} \frac{1}{(\ln(2^{2k}))^2} \\
&\geq \frac{24(\ln(2))^2}{\pi^2} \frac{1}{(\ln(i + 2^{2k}))^2} \\
&= c_i g(i + 2^{2k})
\end{aligned}$$

So, error($ext(h_i), ext(h_j)) \geq c_i g(j)$ for all $j > m_i$, as $i + 2^{2k} \leq j < i + 2^{2(k+1)}$ partitions the cases of interest and g is strictly decreasing.

Second, suppose that h_i contains one or more clauses with a variable symbol. Let $m = \min\{l | P(s^l(X)) \in h_i\}$ and let $h_{i'}$ be obtained from h_i by removing all the clauses with a variable and all the clauses of the form $P(s^l(0))$, where $l \geq m$. Formally:

$$i' = i - \sum_{k \in \{l | P(s^l(X)) \in h_i\}} 2^{2k+1} - \sum_{k \in \{l | l \geq m \wedge P(s^l(0)) \in h_i\}} 2^{2k}$$

To simplify the notation let $f(k) = 2^{2m} + 2^{2(m+1)} + 2^{2(m+1)} + \cdots + 2^{2(k-1)} + 2^{2(k+1)+1}$. Let $m_i = i' + f(\lceil \frac{\lceil \lg(i) \rceil + 1}{2} \rceil) - 1$ and $c_i = \frac{6(\ln(2))^2}{\pi^2}$. The *error* between h_i and h_j, where j is within the interval $i' + f(k) \leq j < i' + f(k+1)$ and $k \geq \lceil \frac{\lceil \lg(i) \rceil + 1}{2} \rceil$, will be either null or at least equal to $D_X(P(s^{k+1}(0))) = \frac{6}{\pi^2 (k+1)^2}$. We may now calculate a bound for this *error* for this range of j.

$$\begin{aligned}
\text{error}(ext(h_i), ext(h_j)) &\geq \frac{6}{\pi^2 (k+1)^2} \\
&\geq \frac{6}{4\pi^2 k^2} \\
&= \frac{6(\ln(2))^2}{\pi^2} \frac{1}{4k^2 (\ln(2))^2} \\
&\geq \frac{6(\ln(2))^2}{\pi^2} \frac{1}{(4k^2 + 12k + 9)(\ln(2))^2} \\
&= \frac{6(\ln(2))^2}{\pi^2} \frac{1}{(\ln(2^{2(k+1)+1}))^2} \\
&\geq \frac{6(\ln(2))^2}{\pi^2} \frac{1}{(\ln(i' + f(k)))^2} \\
&= c_i g(i' + f(k))
\end{aligned}$$

So error($ext(h_i), ext(h_j)) \geq c_i g(i' + f(k))$ for $j \in [i' + f(k), i' + f(k+1))$. As g is a strictly decreasing function, error($ext(h_i), ext(h_j)) \geq c_i g(j)$ for $j \in [i' + f(k), i' + f(k+1))$. Moreover, since $[i' + f(k), i' + f(k+1))$ for $k \geq \lceil \frac{\lceil \lg(i) \rceil + 1}{2} \rceil$ partitions $j > m_i$, error($ext(h_i), ext(h_j)) \geq c_i g(j)$ for $j > m_i$.

Finally, as the hypothesis space is *g-sparse*, Theorem 7.2.2 implies that this concept may be stochastically identified in the limit, with respect to H_{nums}, by Algorithm 12. ■

We now address learnability of concepts definable in an ILP setting. Note that, we do not explicitly include background knowledge in the following theorem, although, it may implicitly be part of the hypothesis space.

Theorem 7.3.4 *Let $\mathcal{L}$ be a first-order language, where P is the target predicate symbol. Let X be the instance space consisting of all the ground atoms that may be formed using P and the ground terms from the Herbrand universe of $\mathcal{L}$. Let H_C be a hypothesis space consisting of clausal logic programs, such that H_C is uniformly decidable. Then the class of concepts definable from H_C, denoted C, are stochastically learnable in the limit with respect to some hypothesis space H_R which is constructed from H_C.*

PROOF: We initially prove the theorem assuming that we have access to an algorithm that can compute the *error* between the extensions of any two hypotheses of H_C. Later on, we show how this assumption is not needed. We take this approach because it helps illustrate and motivate the proof.

The proof works by first constructing a concept class S along with a hypothesis space H_S such that S is a subset of the concept class defined in the theorem. The hypothesis space H_S is shown to be *g-sparse*. Finally, a hypothesis space H_R is generated by combining H_S and H_C, this new hypothesis space is shown to be complete for C and *g-sparse*, thus, C is stochastically learnable in the limit.

Let P be the target predicate for the learner, where P is a predicate symbol of $\mathcal{L}$. [6] Let $X = \{p_1, p_2, \dots\}$ be the set of all ground atoms for the target predicate P given the Herbrand universe of $\mathcal{L}$. We assume $D_X(p_i) = \frac{s_1}{i^{3/2}}$ where s_1 is the normalizing constant.

Let the concept class S be X and $\emptyset$ together with $\{\{p_1, p_2, \cdots, p_k\} | k \in \mathbb{N}\}$. Let $H_S = \{h_{s1}, h_{s2}, h_{s3}, \dots\}$ where h_{s1} is the logic program consisting of the single clause:

$$P(V_1, V_2, \dots V_n).$$

And h_{s2} is the empty logic program. Also h_{si} is the logic program consisting of the clauses:

$$\begin{gathered} p_1. \\ p_2. \\ \vdots \\ p_{(i-2)}. \end{gathered}$$

This concept class along with the hypothesis space and instance space is isomorphic to that of Theorem 7.3.2. Hence, it is also *g-sparse* where $g(i) = 1/i^{1/2}$.

We now combine H_C and H_S to form H_R. This is achieved by Algorithm 13.

It remains to show that H_R is both *g-sparse* and is complete for C. Clearly, H_R is *g-sparse* as hypotheses are only placed in H_R when the *g-sparse* condition is asserted. H_R is complete for C as the algorithm never stops adding hypothesis from H_C, this may be shown by contradiction. Suppose algorithm 13 stopped its output of hypotheses from H_C then there is an index i_C which it has stopped at. And from this point on the algorithm outputs hypothesis from H_S. Now the minimum *error* between h_{ci_C} and the previous hypothesis of H_C will be $\nu_C > 0$; hence at some point $g(i_R) < \nu_C$. So the *error* between h_{ci_C} and the previous hypothesis of H_R must be decreasing to zero because of the hypotheses form H_S being added to H_R. This is impossible because $\text{error}(\mathit{ext}(h_{ci_C}), \mathit{ext}(h_{s1})) > 0$ and $\lim_{i\to\infty} \text{error}(\mathit{ext}(h_{si}), \mathit{ext}(h_{s1})) = 0$; hence:

$$\lim_{i\to\infty} \text{error}(\mathit{ext}(h_{ci_C}), \mathit{ext}(h_{si})) = \text{error}(\mathit{ext}(h_{ci_C}), \mathit{ext}(h_{s1}))$$

with this nonzero limit the *error* between the extension of h_{ci_C} and the hypothesis added from H_S cannot decrease to zero. This proves C is stochastically learnable in the limit.

Note that, the above algorithm requires the calculation of the *error* between extensions of two hypotheses which, in general, is uncomputable. We now show how this assumption can be removed. The approach is similar to the problem of calculating $\theta(\mathit{ext}(h))$. As the hypothesis

[6] Note the proof may be easily extended to the multiple predicate learning setting.

Input :

An indexed hypothesis space H_C and H_S.

A instance space X and distribution D_X.

Output :

An indexed hypothesis space H_R.

$i_C := 1$

$i_S := 1$

$i_R := 1$

repeat

$\nu := \min\{\text{error}(\mathit{ext}(h_{ri}), \mathit{ext}(h_{ci_C}))|i \in \{1 \ldots (i_R - 1)\}$

if $\nu = 0$ or $\nu > g(i_R)$ **then**

$h_{ri_R} := h_{ci_C}$

$i_C := i_C + 1$

else

$h_{ri_R} := h_{si_S}$

$i_S := i_S + 1$

fi

output h_{ri_R}

$i_R := i_R + 1$

forever

Algorithm 13: Process for constructing a *g-sparse* complete hypothesis space for concept class C (assuming we have the *error* function)

space is uniformly decidable and we have an enumeration of the instance space which is non-increasing with respect to the instance space distribution, we can evaluate both $\theta(\mathit{ext}(h))$ and $\text{error}(\mathit{ext}(h_1), \mathit{ext}(h_2))$ to any level of accuracy. Let $\gamma > 0$ be the required accuracy. Then there exists a computer program that will compute $\text{error}_\gamma(\mathit{ext}(h_1), \mathit{ext}(h_2))$ such that:

$$\text{error}_\gamma(\mathit{ext}(h_1), \mathit{ext}(h_2)) \leq \text{error}(\mathit{ext}(h_1), \mathit{ext}(h_2)) \leq \text{error}_\gamma(\mathit{ext}(h_1), \mathit{ext}(h_2)) + \gamma$$

. This estimation of the *error* may be achieved by summing the instance space distribution value over the enumeration of the instance space where h_1 and h_2 disagree on the instance in question. This summing continues until $1 - \gamma$ of the instance space is checked. Another important fact to consider when estimating the *error* using this approach is:

$$\text{error}(\mathit{ext}(h_1), \mathit{ext}(h_2)) = 0 \Rightarrow \text{error}_\gamma(\mathit{ext}(h_1), \mathit{ext}(h_2)) = 0$$

. We may now modify Algorithm 13 such that it returns a *g-sparse* hypothesis space even if the *error* function is not exact.

Algorithm 14 is very similar to Algorithm 13. The main difficulty arises when $\nu = 0$, for it is unknown if $\min\{\text{error}(\mathit{ext}(h_{ri}), \mathit{ext}(h_{ci_C}))|i \in \{1 \ldots (i_R - 1)\} = 0$. We only know that it is somewhere between 0 and γ. So h_{ci_C} is placed in T and we move on. If at any point we find a hypothesis in T that we know has nonzero error with respect to hypotheses output so far, then the algorithm outputs form H_S until these elements in T are removed. Note, if a hypothesis in

Input :

  An indexed hypothesis space H_C and H_S.
  A instance space X and distribution D_X.

Output :

  An indexed hypothesis space H_R.

$h_{r1} := h_{s1}$
output h_{r1}
$T := \emptyset$, $i_C := 1$, $i_S := 2$, and $i_R := 2$
repeat
  $\gamma := \frac{1}{2} g(i_R)$
  if T is not empty **then**
    $h_T := \text{argmax}_{h \in T}\{\min\{\text{error}_\gamma(ext(h_{ri}), ext(h))|i \in \{1 \ldots (i_R - 1)\}\}$
    $w_T := \min\{\text{error}_\gamma(ext(h_{ri}), ext(h_T))|i \in \{1 \ldots (i_R - 1)\}\}$
  else
    $w_T := 0$
  fi
  if $w_T > g(i_R)$ **then**
    $h_{ri_R} := h_T$
    $T := T - \{h_T\}$
  else
    $\nu := \min\{\text{error}_\gamma(ext(h_{ri}), ext(h_{ci_C}))|i \in \{1 \ldots (i_R - 1)\}$
    if $\nu = 0$ **then**
      $T := T \cup \{h_{ci_C}\}$
      $i_C := i_C + 1$
      $h_{ri_R} := h_{si_S}$
      $i_S := i_S + 1$
    else if $\nu > g(i_R)$ and $w_T = 0$ **then**
      $h_{ri_R} := h_{ci_C}$
      $i_C := i_C + 1$
    else
      $h_{ri_R} := h_{si_S}$
      $i_S := i_S + 1$
    fi
  fi
  output h_{ri_R}
  $i_R := i_R + 1$
forever

Algorithm 14: Constructing a *g-sparse* complete hypothesis space for the concept class C

T has zero error with respect to any hypothesis in the output list then it will never be removed from T, this is not a problem as the concept represented by the excluded hypothesis already has a representative in H_R, namely the hypothesis which excludes it.

Now the resulting hypothesis space H_R is *g-sparse* as hypotheses are only placed in H_R when the *g-sparse* condition is asserted.

■

7.4 A Modification of the Q-heuristic and the Learning Algorithm

Until now we have analysed the *Q-heuristic* by directly applying it within the framework of Algorithm 12. This is a very strict analysis. However, if we modify both the *Q-heuristic* and how it is applied in Algorithm 12, making it similar to the approach taken in Laird's [56] thesis, it is possible to disregard the *g-sparse* constraint, and show that the modified approach learns a more general class. The modified approach, however, has the drawback of requiring vast amounts of data for learning. Also, the approach does not translate nicely for a batch learner. Effectively we are modifying the learner to pass the assessment criterion, as opposed to using the criterion to directly assess the learner. We now explain the modification to the *Q-heuristic* and then outline the learning algorithm. Note that the details of the proof are left out, as they are not central to this thesis, rather an overview of the proof is given.

First, the *Q-heuristic* is modified as follows:

$$\begin{aligned} Q'_\sigma(h) = & |\mathrm{TP}_h| \lg\left(\frac{1-\epsilon}{w/2 + (1-w)\theta(ext(h))} + \epsilon\right) + \\ & |\mathrm{TN}_h| \lg\left(\frac{1-\epsilon}{w/2 + (1-w)(1-\theta(ext(h)))} + \epsilon\right) + |\mathrm{FP}_h| \lg(\epsilon) + |\mathrm{FN}_h| \lg(\epsilon) \end{aligned}$$

Where $w = 1/10$. This changes two aspects of the *Q-heuristic*:

1. the *a prior* component is removed and
2. clear bounds are placed on the range the random variable attached to each example may take on.

The introduction of w also places a restriction on the level of noise a system using this heuristic will learn under. That is, $\mu \leq \epsilon \leq 1 - 2w$. Note, when the expected value for the difference between the Q' values of a target and incorrect hypothesis is calculated, this expected value is positive. This may be shown in an analogous way to that of Lemma B.0.1.

Second, we use Algorithm 15 which is essentially the same as Algorithm 5.10 of Laird's thesis [56]. A queue data structure is used in this algorithm[7]. Let $d_{\max}$ be the maximum possible range of the random variables Z_i. This algorithm could fail in one of two ways: by converging to an incorrect hypothesis, or by not converging at all. These are now both shown to be impossible.

[7] A queue $S = \langle e_1, e_2, \cdots e_n \rangle$ is a list of elements. Operations on a queue include: concatenation $\langle a_1, a_2, \cdots a_n \rangle + \langle b_1, b_2, \cdots b_m \rangle = \langle a_1, a_2, \cdots a_n, b_1, b_2, \cdots b_m \rangle$, element selection $\langle e_1, e_2, \cdots e_n \rangle(i) = e_i$, queue rotation $\mathrm{rotate}(\langle e_1, e_2, \cdots e_n \rangle) = \langle e_2, e_3, \cdots e_n, e_1 \rangle$, and queue size evaluation $|\langle e_1, e_2, \cdots e_n \rangle| = n$.

Input :

  A hypothesis space $H = \{h_1, h_2, h_3, \dots\}$.

  An example text E.

Output :

  An infinite sequence of hypotheses.

$S := \langle\rangle$

$r := 1$

repeat

  $I_r := \sqrt{\frac{\lg r}{r}}$

  $m_r := \frac{d_{\max}^2 \ln(r2^r)}{2I_r^2}$

  $S := S + \langle h_r \rangle$

  $\nu := \max\{Q'_{E[m_r]}(S(i)) | i \in \{1 \dots |S|\}\}$

  if $\nu > Q'_{E[m_r]}(S(1)) + I_r$ **then**

    repeat

      $S := \mathrm{rotate}(S)$

    until $\nu \leq Q'_{E[m_r]}(S(1)) + I_r$

  fi

  output $S(1)$

  $r := r + 1$

forever

Algorithm 15: Stochastic identification without the *g-sparse* constraint using an approach similar to Laird [56].

First, suppose the hypothesis converges to h_δ where δ is the error between the extension of h_δ and the target concept C. As the algorithm converges to h_δ at some point, h_δ will remain at the front of the queue forever. If we consider some round r_0 after a target hypothesis h_t has been enqueued, the algorithm has converged to h_δ, and I_r is sufficiently small such that $I_r < \frac{p\beta_p + n\beta_n}{2}$. Using a similar approach to that of the bound described in Section 7.2.1 (Page 115) it is possible to find a bound on the probability h_δ is not displaced by h_t in round r, that is:

$$\begin{aligned}\Pr\{\overline{Z} - E(\overline{Z}) \leq -(\frac{p\beta_p + n\beta_n}{2m})\} &\leq e^{-(\frac{\ln(2)\beta^2 d_{\max}^2}{4d_{\max}^2}(r + \frac{r^2 \ln(2)}{\ln(r)}))} \\ &\leq e^{-(\frac{\ln(2)\beta^2}{4} r)}\end{aligned}$$

This series converges, and hence, the learner will not converge to an incorrect hypothesis.

Second, the algorithm may fail if it continually changes its mind; never converging to any hypothesis. If this is the case then a correct hypothesis must be continually ejected from the front of the queue. If we consider rounds after a correct hypothesis is enqueued then we can calculate a bound on the probability a correct hypothesis, which is at the front of the queue, will be ejected. Note, the hypothesis is ejected if $Q'_{E[m_r]}(S(i)) > Q'_{E[m_r]}(S(1)) + I_r$ for some i. The probability this occurs for a particular i may be bound as follows:

$$\begin{aligned}\Pr\{\overline{Z} < -I_r\} &\leq \Pr\{\overline{Z} - E(\overline{Z}) \leq -I_r\} \\ &\leq e^{\frac{-2m_r(I_r)^2}{d_{\max}^2}} \\ &\leq e^{-\ln(r2^r)} \\ &= \frac{1}{r2^r}\end{aligned}$$

Since there are at most r hypotheses on the queue that could potentially challenge the correct hypothesis at the front of the queue, the displacement event is bound by $\frac{1}{2^r}$. This series converges. Hence, with probability 1 a correct hypothesis will be evacuated from the front at most finite number of times.

Chapter 8

Boosting

8.1 Introduction

Boosting is a technique, introduced by Schapire [117], that builds a composite classifier by iterating an existing learning system. Within the PAC framework it is shown to transform a "weak" learner into a "strong" learner. The type of boosting used in this investigation is *Adaboost.M1*, which extends majority voting to a weighted majority. *Adaboost.M1* was developed by Freund and Schapire [40, 39]. The classifier induced is composed of a set of weighted classifiers which vote to give a classification of an instance. By using propositional learners as the base classifier, boosting induces a composite classifier which in many cases outperforms, in terms of predictive accuracy, the best propositional classifier. An example of this is the evaluation of Quinlan's C4.5 [106] on a representative collection of datasets from the UCI Machine Learning Repository [71]. Quinlan also demonstrates boosting's effectiveness in ILP using his FFOIL system [106].

However, boosting does not always improve predictive accuracy. In fact it is only guaranteed to improve the accuracy on the training data when the base learner produces classifiers that perform better than 50% on the weighted training data. In some situations this improved accuracy on the training data results in poor predictive accuracy on unseen data. This reduced performance generally occurs when: the base learning system is very stable, there is reasonable level of noise in the data, or if the base learner exactly learns the concept represented by the training data.

Boosting effectively enhances the learner in two ways. First, it reduces the variance in the classifier. If the base learning system is unstable, i.e., small changes in the training data produce a change in the classifier induced, then the classifier induced by the boosted learner will be an average classifier over many classifiers and hence be considerably more stable. This is similar to the variance reduction caused by bagging [22]. Second, boosting generally extends the hypothesis space by representing the classifier as a linear combination of the base classifier. This linear combination is used to "home in" on the training data reducing the training error to zero. Hence, the boosted learner will generally learn a larger class of concepts. This partly ignores *a prior* bias and in some cases may worsen the predictive accuracy of the boosted classifier. For example, if there is noise in the training data then the boosted classifier will "home in" to include this noise, which will "warp" the classifier and hence result in a lower predictive accuracy. This effect does not emerge in bagging, as the learner does not "home in" on a hypothesis that exactly represents the training data; hence, the more consistent results for bagging. Although, bagging will not do as well in some cases, it also will not worsen the predictive accuracy as boosting does in some cases.

Freund and Schapire [40] have proven that if the learning system can always produce a classifier which gives better than 50% accuracy on the training data, then the error on the training data reduces exponentially in the number of trials used. This makes the prospect of applying boosting to ILP, which has only two classes, very attractive, as most ILP systems can easily perform better than 50%. However, by using boosting the hypothesis space is extended and *priors* are effectively ignored by the composite classifier. This will be detrimental to the predictive accuracy if there is noise in the data. Also, if the concept can be represented exactly by the clauses in the background knowledge, boosting does not yield any advantage. However, if the concept is not exactly representable in clausal logic boosting offers an effective method for extending and rounding off the edges in the hypothesis space.

Section 8.2 describes and illustrates the boosting algorithm. Section 8.3 discusses some of the issues where boosting is applied to ILP. In Section 8.4, boosting is incorporated into the LIME system. The effect of noise on boosting is investigated in Section 8.5. Then in Section 8.6 we give the results of an empirical study comparing LIME with and without boosting in a variety of domains. Finally, Section 8.7 concludes and discusses this chapter.

8.2 Description

A classifier C is a mapping from the set of possible instances X to the set of class labels L. $C(x)$ denotes the label for x given by the classifier C. Some language is used to describe a classifier; this places a restriction on the mapping the classifier may have, and defines a class of possible classifiers. A learning system induces a classifier from a set of labelled instances which are weighted. Now, a composite classifier C^* is also a classifier, but consists of a multi-set of weighted classifiers $\{\langle C_1, v_1\rangle, \langle C_2, v_2\rangle, \ldots \langle C_n, v_n\rangle\}$. These classifiers vote to define the mapping of the classifier. So the mapping of a composite classifier is defined as -

$$C^*(x) = \text{argmax}_{l \in L}\left(\sum_{\{i \in \{1\ldots n\} | C_i(x) = l\}} v_i\right)$$

Note, that the class of composite classifiers will in general be larger than the class of classifiers used in the composite classifier.

Boosting is a technique which constructs a composite classifier from a base classifier produced by an existing learning system. It is an iterative process where the objective is to produce a classifier that perfectly classifies the training examples. Boosting achieves this by giving a higher weight to the training instances that are incorrectly classified in the previous iteration. This forces the learner to correctly classify these training instances.

Let $E = \{(x_1, l_1), (x_2, l_2), \ldots (x_n, l_n)\}$ be the labelled training data. Boosting then uses T iterations to induce the final result. Initially, a uniform weight vector is associated with the training data $\forall i \in \{1 \ldots n\} w_i^1 = \frac{1}{n}$. The process for $k = 1, 2, \ldots, T$ proceeds as follows: in iteration k both E and w^k are given to the base learning system which induces the classifier C_k. The error of C_k is calculated with respect to w^k so $e_k = \sum_{\{i | C_k(x_i) \neq l_i\}} w_i^k$. From e_k the weight vector for the next iteration is calculated:

$$w_i^{(k+1)} = \begin{cases} w_i^k \frac{1}{2e_k} & C_k(x_i) \neq l_i \\ w_i^k \frac{1}{2(1-e_k)} & C_k(x_i) = l_i \end{cases}$$

This redistributes half of the total weight to the incorrectly classified instances. The voting weight for classifier C_k is also calculated from e_k, $v_k = \log(\frac{1-e_k}{e_k})$, giving more voting power to classifiers with lower error. This completes iteration k and once all T iterations are complete, the composite classifier C^* is induced.

To illustrate the boosting algorithm a basic step-by-step example of how boosting constructs a composite classifier is now given. Suppose the instance space consists of natural numbers between and including 1 and 9. Also, let us suppose that there are exactly two classes + and −. An example is a pair consisting of an instance and its label. Now the base learner induces classifiers, from a multi-set of weighted examples, that may only classify a single segment of the instance space. That is, the segment induced classifies all instances within the segment as + and all instances outside the segment as −. The segment induced will have a minimum error with respect to the weighted examples. This classifier is diagrammatically shown, in Figure 8.1, as a solid bar. The training examples are $\{(1,-),(3,+),(4,+),(5,-),(6,+),(7,+),(8,+),(9,-)\}$. The number of iterations in this example is $T = 3$, generally this would be larger. Initially, the weight of each example is set uniformly at $\frac{1}{8}$, as there are exactly 8 examples. The classifier induced by the base learning system in the first iteration classifies $3, 4, \ldots, 8$ as + and $1, 2, 9$ as −. This incorrectly classifies one example, namely $(5, -)$, giving an error of $\frac{1}{8}$ for the first classifier and a voting weight of 0.845. The weights on the examples are readjusted, increasing the weight on the misclassified example. The process is repeated twice, inducing a composite classifier that classifies $\{3,4,6,7,8\}$ as + and $\{1,2,5,9\}$ as −. Note, this composite classifier correctly classifies the entire training set, outperforming any of the base classifiers. If the number of iterations were increased, the example weight vector would continue to change and so would the induced base classifier. However, the final composite classifier would classify the instance identically to the classifier produced when $T = 3$.

In this basic example if there was no noise in the training examples then the hypothesis space of the base learner, that is single segments, is not sufficient to represent the target concept. However, boosting expands the hypothesis space to allow the concepts to be represented exactly. Although, if $(5, -)$ is a noisy example and the target concept is just a single segment from 3 to 8, then boosting "warps" the induced hypothesis away from the target concept.

8.3 Boosting and ILP

Boosting may easily be adapted for ILP, as the logic programs induced by ILP systems are essentially classifiers with two classes. Quinlan in [106] applies boosting to FFOIL, an ILP system, empirically showing the benefit of boosting for FFOIL. However, in comparison to the propositional setting, the improvement does not turn out to be as good.

Many ILP systems do not have an option enabling the input of weights associated with the training examples. This obstacle is overcome by randomly sampling, with respect to the weight

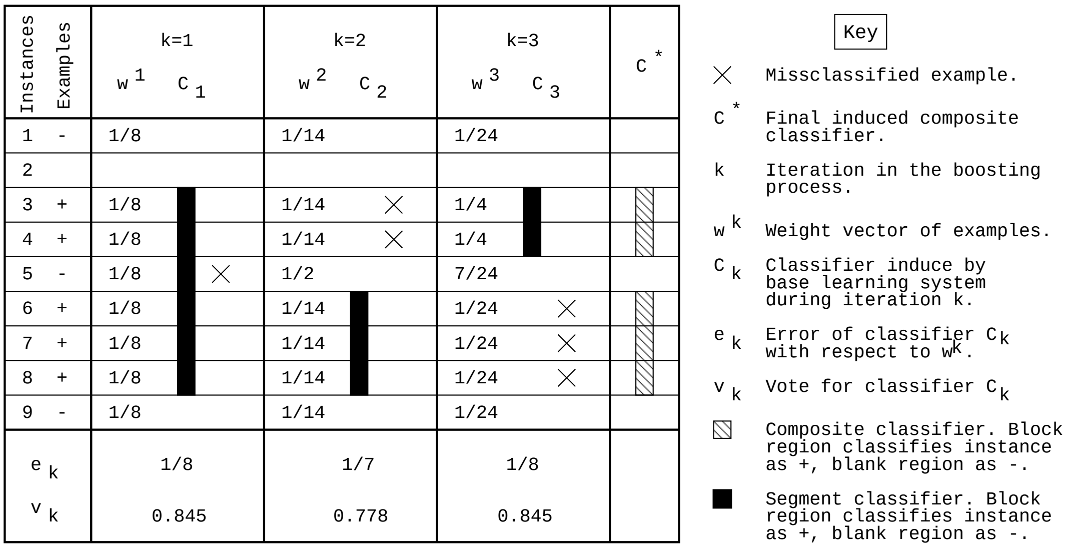


Figure 8.1: Example illustrating the boosting process.

vector, the training examples for each iteration. The training set will reflect the distribution given by the weight vector. Hence, the proportion of training examples the classifier induced incorrectly, will reflect the error with respect to the weight vector. Although, this approach may fail if the learner is not concerned with repetition in the training examples.

8.4 Lime

Boosting was incorporated into to the Lime system to investigate the empirical benefits of boosting in a first-order setting. Rather than invoking Lime within the boosting loop, boosting was incorporated into the Lime system as an option. This is essentially for efficiency reasons, as the pre-processing of background knowledge and the construction of the simple clause table are independent of any weight vector associated with the training examples. Hence, if Lime was called within the boosting loop then these two stages would be redundantly repeated. So, the boosting loop resides within Lime, after the construction of simple clauses and contains both the construction of clauses and induction of a logic program. Clearly, the resulting hypothesis of this boosted Lime is no longer a logic program, but a set of logic programs each voting on the classification of an instance. A diagram of the stages in boosted Lime is shown in Figure 8.2.

A small modification is required to the implementation of the Q heuristic in Lime. As each training example is weighted, the value of $|TP|, |TN|$, and $|FNP|$ must reflect these weights. So these values are replaced appropriately with w_{TP}, w_{TN}, and w_{FNP} respectively, where they are calculated by summing the weights of examples within the set in question. This directs and shapes the induced hypothesis by giving larger Q values to the hypothesis that correctly classifies examples with high weights. Note, the prior and the theta estimation remains unaffected by this modification to the Q heuristic.

The number of boosting trials is set to 10 by default. This is the only parameter used in the boosting process. It is not a critical parameter for the learning system, although, if it is set too

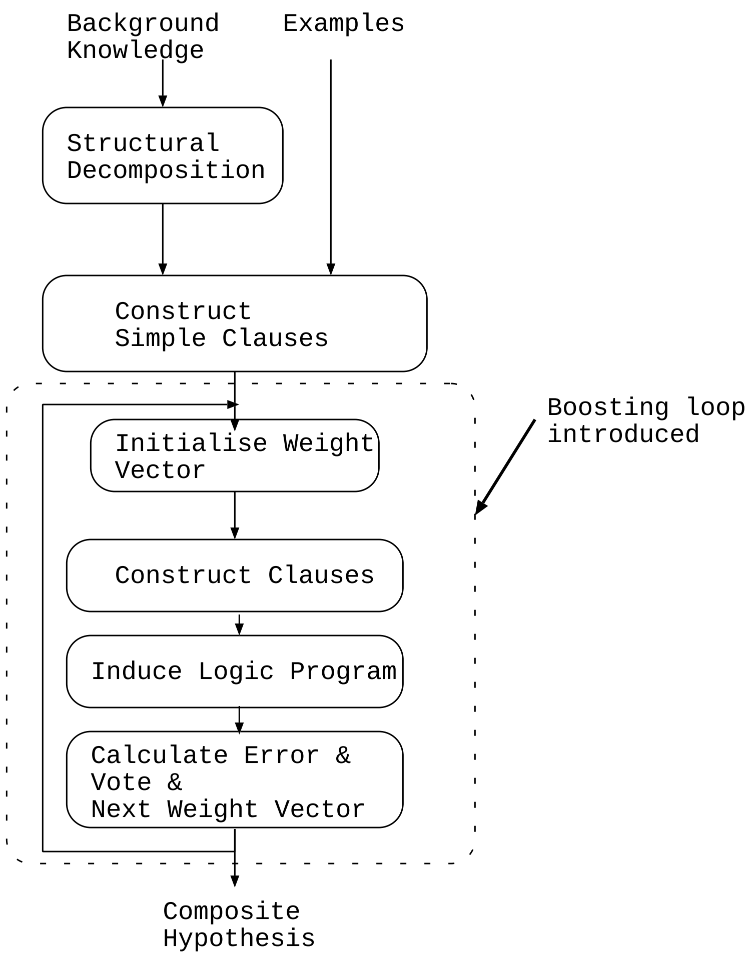


Figure 8.2: Overview of stages of boosted Lime.

high the induction process may take too long. Alternatively, if it is set too low, there may not be enough trials. This is especially the case when the hypothesis induced have weighted errors close to 0.5.

8.5 Noise

When the training examples contain noise, boosting is generally detrimental to the predictive performance of the induced composite hypothesis. This is an important factor to consider when determining whether to apply boosting. If the training examples contain noise it may be more appropriate to use a technique such as bagging or just the base learner rather than using boosting. Boosting reduces the variance in the hypotheses induced by the base learning system. It also extends the hypothesis space, forcing, if it is able, the induced composite classifier to exactly classify the training examples. So, the induced composite classifier will label the noisy examples with a class label corresponding to the label given in the training examples. This effectively "warps" the composite classifier to include, where possible, the noisy examples. Hence, one would expect the error in the composite hypothesis would at best correspond to the level of noise in the training examples.

Let X be the instance space and let D_X be the probability measure over the instance space. let k be the number of classes with labels $\{1 \ldots k\}$. A concept is simply a mapping from the instance space to a class label. Suppose the training examples consist of n examples which are constructed from either noisy or noise free examples. A noisy example is constructed by randomly selecting an instance from X by D_X and then randomly labelling it from $\{1 \ldots k\}$ using a uniform distribution. Nonnoisy examples are constructed by again randomly selecting from X and then labelling using the underlying concept. μ is the proportion of noisy examples in these training examples. Let us assume that the boosting trials always induce classifiers with errors less then 0.5 and there are enough boosting trials to reduce the error on the training examples to zero. The composite classifier will correctly classify all the training examples. From the error in classifying the incorrectly labelled noisy examples we have a lower bound on the error in the composite classifier. This may be calculated by:

$$e_{\mathrm{lb}} = \sum_{\{i \in X \mid (i,l) \in E \& c(i) \neq l\}} D_X(i)$$

However, this may be a very weak bound if the instance space distribution is sparse.

Another way of gaining an estimate of the error in the composite classifier induced by boosting is to use the training set itself to estimate the expected error. The main drawback of this approach is that the induced classifier is not independent of the test set. However, one expects that the proportion of error introduced by the training data into the composite classifier will also translate into the entire space. This is later confirmed by empirical results. So, to estimate the expected error in a composite classifier using the training data, we simply estimate the expected number of incorrectly classified examples. The proportion incorrectly classified is $\mu(\frac{k-1}{k})$. The reason for this is that the composite classifier will correctly classify the entire training set and as μ is the proportion of noisy examples of which $(\frac{k-1}{k})$ are expected to be labelled differently to the labelling given by the target concept. So even if the base learning system is able to ignore the noise in the training examples, boosting will override any advantage gained.

For a real world domain where the exact target concept is unknown and the number of noisy examples in the data gathered is also unknown, the technique of cross-validation is generally used. In this technique, examples are repeatedly partitioned into different training and test sets, and experiments are performed and tested. An average accuracy is taken across all the experiments. Suppose the proportion of misclassified examples is ν and the target concept is representable in the hypothesis space. Furthermore suppose the learning system is able to induce the target hypothesis from any of the training sets presented to it. Now, if cross-validation is used to assess such a domain then the expected accuracy will be $1 - \nu$. Because the hypothesis induced is the target hypothesis that exactly represents the target concept and the test data is expected to contain ν misclassified examples. Now if boosting is used, the proportion expected to be incorrectly classified by the classifier induced by boosting, with respect to the concept, is ν. Now, when this classifier is assessed, the test set is expected to already contain ν noisy examples. The predictive accuracy will only worsen when the boosted classifier is assessed against the test set, as we would expect this noise to be independent. The final expected error given by cross-validation will be $2\nu - (2 - \frac{k-2}{k-1})\nu^2$. Hence, the predictive error will almost double when a learning system that handles noise well is boosted and the test data contains the same level of noise as the training data.

To study the effect of noise on boosting, an artificial problem is created. This enables exact control over the level of noise introduced into the training data, so, the boosted and nonboosted results may be compared at different levels of noise. The concept used is the *king-rook-king* illegal move described in Chapter 6. We briefly describe the concept again. This concept is exactly representable in clausal form and given in Table 8.5. The relation *illegal* checks if an end game position is an illegal chess position, where the remaining pieces are the white king, white rook, and black king. The board will be in an illegal state in the following situations: a piece is on top of another piece, the kings are within a square of each other, the black king is on the same file as the white rook where the white king is not in between them, or the black king is on the same rank as the white rook where the white king is not in between them. These are asserted by clauses 1-3, 4, 5-7, and 8-10 respectively.

In each trial the training example set is constructed from 300 examples of which a proportion are noisy. Examples that are not noisy are generated by randomly placing each piece on the board using a uniform distribution. The board "setup" forms the instance which is labelled as a positive example if it is an illegal "setup", otherwise it is labelled as a negative example. A noisy example is constructed by again randomly positioning each piece, then randomly labelling the board "setup" 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 10,000 noise free random examples and calculating the proportion of these the hypothesis correctly classifies. Each trial is repeated 20 times and the mean and standard error of the accuracy is estimated. This is repeated with and without boosting at different noise levels. The results of these experiments are shown in Figure 8.3. The error bars show the standard error. The number of boosting trials used is 50, and the default noise parameter, of 0.1, is used for both boosted and nonboosted LIME.

Without boosting, LIME handles this problem well for noise levels from 0 to 45%, giving an average accuracy of around 97%. At this point LIME's accuracy drops sharply as the hypothesis

$$\begin{array}{l}
illegal(WK_f, WK_r, WR_f, WR_r, BK_f, BK_r) \leftarrow WK_f = WR_f, WK_r = WR_r. \\
illegal(WK_f, WK_r, WR_f, WR_r, BK_f, BK_r) \leftarrow WK_f = BK_f, WK_r = BK_r. \\
illegal(WK_f, WK_r, WR_f, WR_r, BK_f, BK_r) \leftarrow BK_f = WR_f, BK_r = WR_r. \\
illegal(WK_f, WK_r, WR_f, WR_r, BK_f, BK_r) \leftarrow adj(WK_f, BK_f), adj(WK_r, BK_r). \\
illegal(WK_f, WK_r, WR_f, WR_r, BK_f, BK_r) \leftarrow BK_f = WR_f, BK_f \neq WK_f. \\
illegal(WK_f, WK_r, WR_f, WR_r, BK_f, BK_r) \leftarrow \\
\qquad\qquad BK_f = WR_f, BK_f = WK_f, WK_r < BK_r, WK_r < WR_r. \\
illegal(WK_f, WK_r, WR_f, WR_r, BK_f, BK_r) \leftarrow \\
\qquad\qquad BK_f = WR_f, BK_f = WK_f, WK_r > BK_r, WK_r > WR_r. \\
illegal(WK_f, WK_r, WR_f, WR_r, BK_f, BK_r) \leftarrow BK_r = WR_r, BK_r \neq WK_r. \\
illegal(WK_f, WK_r, WR_f, WR_r, BK_f, BK_r) \leftarrow \\
\qquad\qquad BK_r = WR_r, BK_r = WK_r, WK_f < BK_f, WK_f < WR_f. \\
illegal(WK_f, WK_r, WR_f, WR_r, BK_f, BK_r) \leftarrow \\
\qquad\qquad BK_r = WR_r, BK_r = WK_r, WK_f > BK_f, WK_f > WR_f.
\end{array}$$

Table 8.1: Logic for the king-rook-king illegal move.

induced become less reliable. With no noise introduced into the training data boosted LIME performs better than LIME without boosting, giving an accuracy of 99%. This is due to boosting re-weighting the examples, forcing the composite hypothesis to correctly classify some of the examples that come from the cases that only form a small part of the instance space. Whereas LIME, without boosting, uses the Q heuristic to decide that a small hypothesis that misclassifies a few examples is better than a larger one that perfectly classifies all the examples. Hence, this approach induces a hypothesis that does not perform as well as boosted LIME. However, once noise is introduced this disadvantage becomes an advantage, as LIME, without boosting, will see noisy examples as noise and produce a hypothesis that ignores these examples. However, boosted LIME will not ignore these noisy examples but will induce a composite hypothesis that will include them. Hence the classifier's accuracy will be approximately $1 - \frac{\mu}{2}$, where μ is the level of noise. The graph clearly bears this relationship, as the graph of boosted LIME is very close to a straight line from 100% accuracy at 0% noise to 50% accuracy at 100% noise.

These results show that it is advantageous to use boosting in the *k-r-k* domain when there is no noise in the training examples. Yet, when the training examples contain noise, then applying boosting is detrimental to the predictive accuracy of the induced hypotheses. This is generally true across different domains, as the reason it occurs is essentially independent of the domain.

8.6 Empirical Results

Most of the data sets used in this section were obtained from the UCI Machine Learning Repository [71] and this resource is greatfully acknowledged.

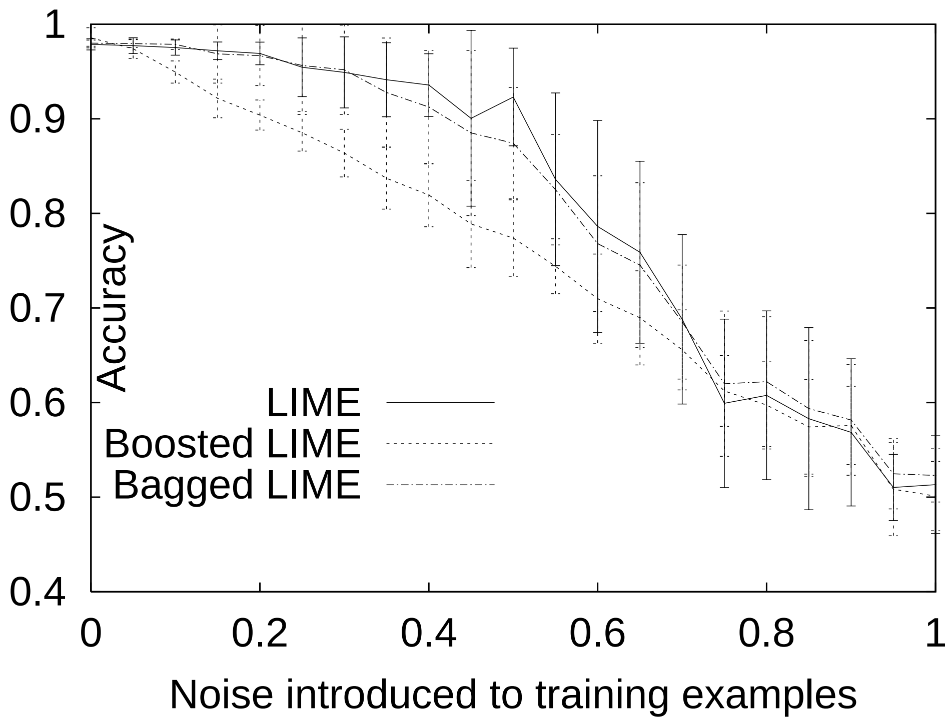


Figure 8.3: Comparison of LIME with and without boosting as noise is introduced using the the king-rook-king example.

Boosting ?	Error	SE
No	3.6%	0.5
Yes	0.4%	0.2

Table 8.2: Results for LIME with and without boosting in the student loan domain.

8.6.1 Student Loan

We refer the reader to Section 6.6.2 (on Page 94) for a description of the concept, training examples, and background knowledge.

A 10-fold cross validation was used to assess the predictive error of LIME both with and without boosting. In the case of LIME without boosting, the average error was 3.6%, with a standard error of 0.5. For boosted LIME the average error was 0.4% with a standard error of 0.2. These results are shown in Table 8.2, an estimation of the standard error is given, rather than the sample standard deviation. Boosting clearly reduces the average error, this reduction is statistically significant at a level of 0.025. The test statistic used assumes values are normally distributed and the standard deviations are unknown and possibly different. The statistical hypothesis test used is described in Appendix A.

In this domain it is clear that boosting is beneficial to the predictive accuracy of the induced hypotheses.

Algorithm	Accuracy
NewID	84.0%
CN2	98.1%
MBRtalk	88.4%
IB1	98.1%
IB3	82.0%
IB3-CI	99.1%

Table 8.3: Results reported for the tic-tac-toe domain in [4]

Boosting ?	Threex in BG ?	Accuracy	SE
No	Yes	100%	0
Yes	Yes	100%	0
No	No	96%	1.13
Yes	No	98%	0.36

Table 8.4: Results for the tic-tac-toe domain with and without boosting.

8.6.2 Tic-Tac-Toe

We refer the reader to Section 6.6.3 (on Page 95) for a description of the concept, training examples, and background knowledge.

The experimental set up used by Aha [4] has been followed so their results may be compared. The examples are randomly partitioned into training and test sets, where the training sets consists of approximately 70% of the examples and the test sets 30%. The learning system induces its hypothesis from the training set and the accuracy of this hypothesis is estimated using the test set. This entire process is repeated 10 times and the average accuracy is calculated.

The results reported by Aha [4] are shown in Table 8.3. These are all propositional learners which find this concept more difficult to represent than representing the concept in clausal form. In fact, when LIME is given the background relation "three x's", it has no trouble inducing a hypothesis which exactly represents the concept, as the concept is representable with eight clauses each of which contain one literal in their bodies. However, without this background knowledge LIME has more difficulty. Although the target hypothesis consists of the same eight clauses each clause now consists of three literals in their bodies. LIME is only able to exactly represent the concept 4 out of the 10 trials. When a base learner induces a hypothesis that exactly represents the concept and there is no noise in the training examples, boosting will have no effect as the boosting loop will halt after one iteration and return the same hypothesis that would be induced without boosting. Hence, with "three x's" in the background knowledge, the result for boosting is exactly the same without boosting and of no great interest. However, without this background knowledge, boosting improves the accuracy of the hypothesis induced. This assertion is statistically tested at a level of 0.10. These results are shown in Table 8.4.

This domain has also been studied in [67] and [66]. However, as the experimental set is different there is no point comparing results.

System	Boosting ?	Error	SE
LIME	No	4.41%	0.76
LIME	Yes	4.11%	0.68
LIME (-n 3)	No	3.38%	0.76
LIME (-n 3)	Yes	3.52%	0.82
FFOIL	No	5.2%	-
FFOIL	Yes	4.0%	-
c4.5	No	5.3%	-
c4.5	Yes	4.0%	-

Table 8.5: Results for breast cancer domain.

8.6.3 Wisconsin breast cancer

This domain is considered as it contains no relational information. Thus, we may directly compare the effect of boosting on LIME, an ILP system, and a propositional learner such as c4.5 [105]. We refer the reader to Section 6.6.4 (on Page 97) for a description of the concept, training examples, and background knowledge.

A 10-fold cross-validation is conducted with this data and the averages are calculated. The results for FFOIL and c4.5 are reported in [106], but the sample standard deviations for the sets of errors were not reported. The results are shown in Table 8.5.

Interestingly, all three system give similar results. However, LIME performs a little better than the other systems, without boosting, yet it also performs slightly worse when boosting is applied. If each of the 10 trials are considered individually, then boosting reduces the error in only 4 of the 10 trials, increases the error in 2 of the 10 trials, and the error remains the same in the remaining 4 trials. The average reduction in error when boosting is applied to LIME, is not statistically significant even at a level of 0.10.

The same experiment was repeated, using identical training examples, with the noise parameter set at 3% as opposed to the default 10%. This improved the average error of LIME by one percent. However, the boosted result now has a higher average error than the nonboosted case, although, the boosted result is still better than previous results. Unfortunately, between all the averages of the trials with LIME none of the means are statistically different, this is due to the small number of trials and the relatively high sample standard deviation. To determine if there is a statistically significant difference between boosted and nonboosted LIME, more trials have been performed.

Ten 10-fold cross-validations are performed on the same examples, where these examples are shuffled between each cross-validation giving a total of 100 trials. The results are shown in Table 8.6. Note, the noise parameter is also set at 3% for these trials. Interestingly, there is no significant difference between the nonboosted and boosted results. This also confirms the usefulness of statistical testing as previously the nonboosted average was lower than the boosted case. However, the difference was not statistically convincing. Now, with more trials it is clear that there is no significant difference between the performance of boosted and nonboosted LIME on the Wisconsin breast cancer examples.

System	Boosting ?	Error	SE
LIME (-n 3)	No	4.23%	0.265
LIME (-n 3)	Yes	4.24%	0.224

Table 8.6: Results for breast cancer domain with ten 10-fold cross-validations

System	Boosting ?	Error	SE
LIME	No	39.95%	2.14
LIME	Yes	33.57%	1.50

Table 8.7: Results of protein domain with and without boosting.

8.6.4 Protein

The target concept is the secondary structure of α/α type proteins. The data set was obtained from Oxford Universities Computing Laboratory web page and has been studied by Muggleton, King, and Sternberg [91]. [1] The data consists of 16 proteins having an average length of 127 elements. A comparison of nonboosted and boosted LIME is conducted by a leave-one-out cross-validation on the 16 proteins. The results are shown in Table 8.7. The boosted LIME performs better than nonboosted LIME. This result is statistically significant at a level of 0.025.

8.6.5 Votes

We refer the reader to Section 6.6.5 (on Page 97) for a description of the concept, training examples, and background knowledge.

Initially ten 10-fold cross validations are performed on the examples comparing boosted and nonboosted LIME. The average and sample standard deviation of each of the 100 tests is shown in Table 8.8.

The nonboosted LIME shows a lower average error than boosted LIME. However, the difference in the means is not statistically significant at a level of $\alpha = 0.1$, although, it is very close. The critical value of the t-distribution for $\alpha = 0.1$ with 193 degrees of freedom is approximately 1.30. The t value of 1.15 comes close to this critical region. By simply increasing the number of tests, using 15 10-fold cross validation, nonboosted LIME performs better than boosted LIME at a significant level of 0.1. The results of these experiments are shown in Table 8.9

Another interesting aspect to examine in this domain is a comparison between the errors in the training examples. The nonboosted LIME shows an average error of 4.36% on the training examples, whereas boosted LIME shows an average error on the training examples of 0.49%. This stark difference is due to boosting "warping" the induced hypothesis to attempt to exactly match the training examples. However, this "warping" translates to no advantage in the test data, in fact the performance is reduced in comparison to nonboosted LIME.

[1] However, the Oxford group obtained the data set from the Brookhaven data bank [14].

System	Boosting ?	Error	SE
LIME	No	4.31%	0.28
LIME	Yes	4.81%	0.33

Table 8.8: Results of votes domain with and without boosting using 10 10-fold cross validation.

System	Boosting ?	Error	SE
LIME	No	4.36%	0.23
LIME	Yes	4.76%	0.25

Table 8.9: Results of votes domain with and without boosting using 15 10-fold cross validation.

8.7 Discussion

Although the results are not conclusive they suggest that boosting will improve predictive accuracy. This improvement, however, is not as significant as in the case of propositional learners. Also, boosting performs poorly when noise is introduced into the training data.

A drawback of boosting is that the user loses the ability to comprehend the hypotheses, as they are formed by many logic programs voting. Thus, induction loses its power to explain or help the user understand a concept, thereby restricting the use of the composite hypothesis to prediction. This may not be a problem in many domains, however, it is clearly a disadvantage in others.

Chapter 9

Conclusion

The results presented in this dissertation are now briefly summarized and a short discussion of possible future research directions is given.

9.1 Summary

A framework for induction is proposed where the examples are assumed to be generated stochastically and the number of positive and negative examples are fixed and independent of the target concept. A Bernoulli process is used to introduce noise in the framework that models realistic learning scenarios in a variety of domains. Bayesian inductive reasoning is used to establish a heuristic in this framework for inducing the most probable hypothesis given the training examples. This approach requires a prior probability over the hypothesis space.

A number of propositions about the heuristic, referred to as the Q-heuristic, are shown. We note two of them here. First, if the target concept is "small" with respect to the instance space probability measure, then positive examples are more persuasive than negative ones. The converse is also true. Intuitively, this result is to be expected. For example, in learning a natural language, positive examples of the language are considerably more instructive than negative ones. Second, given the training examples, only a finite number of hypotheses need to be considered to find the maximum Q value out of a possibly infinite set of hypotheses. This bounds the search for any inductive system that adopts the Q-heuristic.

The effectiveness of the Q-heuristic is evaluated both from an empirical and theoretical perspective.

9.1.1 Empirical evaluation of the Q-heuristic: Lime system

The design of the ILP system Lime was described. Lime uses the Q-heuristic for comparing competing hypotheses. The notion of a 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. Also given are detailed algorithms for simple clause construction, clause construction, and logic program construction. Additionally, a technique which allows a single recursive clause to be approximately evaluated during the induction process is given.

Empirical results were presented that reinforce the superior noise handling ability of LIME using the Q-heuristic. The performance of LIME is particularly good when it is learning recursive definitions in the presence of noise. The system has also been tested on some real world domains.

The boosting technique was applied to LIME and studied with regard to artificially noisy domains and some real world domains. The results are not conclusive, yet, they show boosting will generally improve predictive accuracy, although, not as significantly as with propositional learners. Moreover, boosting performs poorly when noise is introduced into the training data.

9.1.2 Theoretical evaluation of the Q-heuristic: learnability results

An attempt has been made to give a preliminary theoretical evaluation of the Q-heuristic. To this end, we have shown that a learner using the Q-heuristic stochastically learns in the limit any finite class of concepts even when noise is present in the training examples. This result is encouraging, because with enough data, there is the expectation that the learner will induce a correct hypothesis. The proof of this result is extended to show that a restricted infinite class of concepts can also be stochastically learnt in the limit. The restriction requires the hypothesis space to be *g-sparse*. We show how, given a first-order language, a *g-sparse* hypothesis space may be generated, thereby showing how such a first-order language may be stochastically learnt in the limit. The Q heuristic along with the learning algorithm is modified, this removes the *g-sparse* restriction and we then show a more general class of concepts to be stochastically learnable in the limit. This is a more general result than Laird's [56] as we employ a more general model of training examples.

9.2 Future Work

There is a variety of directions this research may be extended. The following lists some of the potential avenues:

- One important direction this research may take is to examine LIME's approach in more real world domains. This may require extending LIME to handle terms such as lists, sets, or continuous numbers. The carcinogenesis domain would be one such domain of interest. This was a challenge given to the ILP community by Srinivasan, Muggleton, King, and Sternberg at IJCAI-97 [125].
- The stochastic identification in the limit results in Chapter 7 are preliminary. An open question is whether these results can be extended to take into account complexity issues. This would give some idea of the the expected number of training examples provided to the learner before the correct hypothesis is induced. This would be useful in assessing the feasibility of learning.
- One problem faced when assessing first-order systems using artificial domains is the ability of the assessor to alter the training examples until the system successfully learns the target concept. One approach is to randomly generate both the target concept and the training examples, thus giving a more just benchmark for evaluating a first-order system on artificial domains. We use this approach in some of the experiments of Chapter 6. This form of

assessment and comparison could be generalized to provide a fairer benchmark for first-order systems.

- LIME's hypothesis space is limited to determinate clauses. This restricts the scope of concepts which may be learnt. As representation of some concepts may require nondeterminate clauses, expanding the hypothesis space of LIME would be useful.

- Another limitation of LIME is that the background knowledge must be presented extensionally, that is as ground facts. By allowing intentional background knowledge the system would be simpler to use.

- Although, LIME currently has a rudimentary predicate invention attachment, it would be useful to extend this and study its capability.

- The prior probability of a hypothesis is calculated using a size measure. However, in some cases it would be of value to incorporate other measures into this evaluation. For example, suppose an ILP system is given the concept *sort* to learn. There is a vast array of sorting algorithms, this gives rise to a variety of logic programs that will represent this concept. These include *insertion sort* and *merge sort*. Now in terms of size, *insertion sort* is smaller than *merge sort*. Therefore, *insertion sort* will have a larger prior probability and be selected over *merge sort*. However, in terms of time complexity *merge sort* is more efficient, being $O(n \ln n)$ in comparison to $O(n^2)$. Another avenue of research would be to look at incorporating the time complexity of the hypothesis into the calculation of the prior probability.

Bibliography

[1] M. Adams and V. Guillemin. *Measure Theory and Probability.* Birkhäuser Boston, 1996.

[2] D. Aha, S. Lapointe, C. Ling, and S. Matwin. Learning singly recursive relation from small datasets. In F. Bergadano, L. DeRaedt, S. Matwin, and S. Muggleton, editors, *Proc. of the IJCAI-93 Workshop on Inductive Logic Programming*, pages 47–58, Chambery, France, 1993.

[3] D. Aha, S. Lapointe, C. Ling, and S. Matwin. Learning recursive relations with randomly selected small training sets. In William W. Cohen, editor, *Proc. of the Int. Conf. and Machine Learning*, pages 12–18, New Brunswick, NJ, 1994. Morgan Kaufmann.

[4] D. W. Aha. Incremental constructive induction: An instance-based approach. In Morgan Kaufmann, editor, *Proceedings of the Eighth International Workshop on Machine Learning*, pages 117–121. Evanston, ILL, 1991.

[5] D. Angluin. Identifying languages from stochastic examples. Technical Report TR–614, University of Yale, 1988.

[6] D. Angluin and P. Laird. Learning from noisy examples. *Machine Learning*, 2:343–370, 1987.

[7] D. Applebaum. *Probability and information - An intergrated approach.* Cambridge University Press, 1996.

[8] M. Bain, C. Sammut, A. Sharma, and J. Shepherd. ReDuce: automatic structuring and compression in relational databases. In *MLNet Workshop on Data Mining with Inductive Logic Programming, Bari, Italy*, July 1996. Published electronically: (`http://www.ai.univie.ac.at/ilp_kdd`)

[9] R.B. Banerji. A language for the description of concepts. In *General Systems*, pages 135–141, 1964.

[10] F. Bergadano. The problem of induction and machine learning. In *Proc. Int. Joint Conf on Artificial Intelligence*, pages 1073–1079, 1991. Sydney, Australia.

[11] F. Bergadano and D. Gunetti. An interactive system to learn functional logic programs. In *Machine Learning*, 1994.

[12] F. Bergadano and D. Gunetti. Learning clauses by tracing derivations. In *Proceedings of the Fourth International Workshop on Inductive Logic Programming*, 1994.

[13] F. Bergadano and G. Gunetti. *Inductive Logic Programming: from Machine Learning to Software Engineering*. MIT Press, 1996.

[14] F.C Bernstein, T.F. Koetzle, G.J.B. Williams, E.F. Jr Meyer, M.D. Brice, J.R. rodgers, O. Kennard, T. Shimanouchi, and M. Tasumi. ˙*J. Mol. Biol.*, 112:535–542, 1977.

[15] P. Billingsley. *Probability and Measure*. John Wiley & Sons, 1995.

[16] Max Black. *Problems of Analysis : Philosophical Essays*. Routledge & Kegan Paul, 1954.

[17] L. Blum and M. Blum. Towards a mathematical theory of inductive inference. *Information and Control*, 28, 1975.

[18] T. L. Booth and R. A. Thompson. Applying probability measures to abstract languages. *IEEE Trans. Comput.*, C-22:442–450, 1973.

[19] H. Boström. Covering vs. divide-and-conquer for top-down induction of logic proagrams. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 1194–1200, 1995.

[20] H. Boström and P. Idestam-Almquist. Specialization of logic programs by pruning SLD-trees. In S. Wrobel, editor, *Proceedings of the 4th International Workshop of Inductive Logic Programming*, 1994.

[21] I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, Workingham, 1990. second edition.

[22] L. Breiman. Arcing classifiers. Technical report, Statistics Depatment, University of California Berkeley, 1997. Available at: `(http://www.stat.berkeley.edu/usr/breiman/)`

[23] W. Buntine. Generalized subsumption and its applications to induction and redundancy. *Artificial Intelligence*, 36(2):149–176, 1988.

[24] 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.

[25] R.M. Cameron-Jones and J.R. Quinlan. Efficient top-down induction of logic programs. *SIGART Bulletin*, 5(1):33–42, 1994.

[26] P. Clark and T. Niblett. The CN2 induction algorithm. *Machine Learning*, 3(4):261–283, 1989.

[27] B.L. Cohen. *A theory of structural concept formation and pattern recognition*. PhD thesis, Department of Computer Science, University of New South Wales, 1978.

[28] Congressional quarterly almanac, 98th congress, 2nd session 1984, 1985.

[29] Gamberger. D. and N. Lavrač. Noise detection and elimination applied to noise handling in a KRK chess endgame. In *Proceedings of the 6th International Workshop on Inductive Logic Programming*, pages 59–75, 1996.

[30] L. De Raedt and M. Bruynooghe. An overview of the interactive concept-learner and theory revisor CLINT. In S. Muggleton, editor, *Inductive Logic Programming*, volume 38 of *APIC Series*, pages 163–191. Academic Press, London, 1992.

[31] L. De Raedt and M. Bruynooghe. A theory of clausal discovery. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, San Mateo, CA, 1993.

[32] B. Dolšak. Determining the geometric model of finite element meshes using ai methods. Technical report, University of Maribor: CAD Centre Report, 1991.

[33] B. Dolšak and S. Muggleton. The application of inductive logic programming to finite element mesh design. In *Proceedings of the 1991 Workshop on Inductive Logic Programming, Viana do Castelo, Portugal*, 1991.

[34] S Džeroski. Handling noise in inductive logic programming. Master's thesis, Faculty of Electrical Engineering and Computer Science, University of Ljubljana, Slovenia, 1991.

[35] S. Džeroski 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.

[36] S. Džeroski and B. Dolšak. Comparison of ilp systems on the problem of finite element mesh design. In *Proceedings of the Sixth ISSEK Workshop*, 1992. Jozef Stefan Institute, Ljubljana, Slovenia.

[37] S. Džeroski, 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.

[38] F. Esposito, D. Malerba, and G. Semeraro. Multistrategy learning for document recognition. *Applied Artificial Intelligence*, 8(1):33–84, 1994.

[39] Yoav Freund and Robert E. Schapire. A decision theoretic generalization of on-line learning and an application to boosting. Available at: `(http://www.research.att.com/orgs/ssr/people/{yoav,schapire})`

[40] Yoav Freund and Robert E. Schapire. A decision theoretic generalization of on-line learning and an application to boosting. In *Computational Learning Theory : Second European Conference, EuroCOLT 95*, volume 23-27. Springer-Verlag, 1995.

[41] E. M. Gold. Limiting recursion. *Journal of symbolic Logic*, 30, 1965.

[42] E. M. Gold. Language identification in the limit. *Information and Control*, 10:447–474, 1967.

[43] A. C. Grayling, editor. *Philosophy.* Oxford University Press, 1995.

[44] Marko Grobelnik. Induction of prolog programs with markus. Jožef Stefan Institute, 1994.

[45] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American statistical association*, 58:13–30, 1963.

[46] D. Hume. *Enquire concerning human understanding and concerning the principles of morals.* Oxford University Press, 2nd edition, 1902.

[47] Popplestone R. J. An experiment in automatic induction. *Machine Intelligence*, 5:101–124, 1970.

[48] S. Jain, D. Osherson, J. Royer, and A. Sharma. *Systems That Learn: An Introduction to Learning Theory*. MIT Press, second edition edition, 1999.

[49] 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.

[50] J-U. Kietz and S. Wrobel. Constructive induction : Learning concepts through syntactic and task-oriented models. In S. Muggleton, editor, *Inductive Logic Programming*. Academic Press, London, 1991.

[51] J. U. Kietz and S. Wrobel. Controlling the complexity of learning in logic through syntatic and task-oriented models. In *Proc. Workshop on Inductive Logic Programming*, 1991.

[52] B. Kijsirikul, M. Numao, and M. Shimura. Efficient learning of logic programs with non-determinate, non-discriminating literals. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 417–421, 1991.

[53] B. Kijsirikul, M. Numao, and M. Shimura. Discrimination-based constructive induction of logic programs. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 44–49, San Jose, CA, 1992.

[54] R.D. King, Muggleton S., R. Lewis, and M.J.E. Sternberg. Drug design by machine learning: the use of inductive logic programming to model the structure-activity relatioships of trimethoprim analogues binding to dihydrofolate reductase. In *Proceedings of the 1992 Workshop on Inductive Logic Programming, Tokyo*, 1992.

[55] M. Kirschenbaum and L. Sterling. Refinement strategies for inductive learning of simple prolog programs. In *Proc. of the 9th inter. conference on Artificial Intelligence*, 1991.

[56] P. Laird. *Learning from Good Data and Bad.* PhD thesis, Yale University, 1987.

[57] P. Laird. *Learning from Good and Bad Data.* Kluwer Academic Publishers, Boston, MA, 1988.

[58] S. Lapointe and S. Matwin. Sub-unification : A tool for efficient induction of recursive programs. In *Proceedings of the Ninth International Conference on Machine Learning*, pages 273–281, Aberdeen, Scotland, 1992. Morgan Kaufmann.

[59] N. Lavrač and S. Džeroski. Background knowledge and declarative bias in inductive concept learning. In K. P. Jantke, editor, *Analogical and Inductive Inference*, Lecture Notes in Artificial Intelligence. Springer-Verlag, October 1992.

[60] N. Lavrač and S. Džeroski. Inductive learning of relations from noisy examples. In S Muggleton, editor, *Inductive Logic Programming*, volume 38 of *APIC Series*, pages 495–516. Academic Press, 1992.

[61] N. Lavrač and S. Džeroski. *Inductive Logic Programming: Techniques and Applications.* Ellis Horwood (Simon and Schuster), 1994.

[62] N. Lavrač, S. Džeroski, and I. Bratko. Handling imperfect data in inductive logic programming. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, pages 48–64. IOS Press, 1996.

[63] N. Lavrač, S. Džeroski, and M. Grobelnik. Learning nonrecursive definitions of relations with LINUS. In *Proceedings of the Fifth European Working Session on Learning*, pages 265–281. Springer-Verlag, 1991.

[64] C. X. Ling. Learning the past tense of English verbs: The symbolic pattern associator vs. connectionist models. *Journal of Artificial Intelligence Research*, 1:209–229, 1994.

[65] J.W. Lloyd. *Foundation of Logic Programming.* Springer, New York, 1984.

[66] C. J. Matheus. Adding domain knowledge to sbl through feature construction. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 803–808. AAAI Press, 1990.

[67] C. J. Matheus and L. A. Rendell. Constructive induction on decision trees. In Morgan Kaufmann, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 645–650, 1989.

[68] 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.

[69] 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.

[70] B. Meltzer. Power amplification for theorem-provers. *Machine Intelligence*, 5:153–163, 1970.

[71] C.J. Merz and P.M. Murphy. UCI repository of machine learning databases, 1996.

[72] R. Michalski and R. Stepp. Learning from observation : Conceptual clustering. In R. Michalski, J. Carbonell, and T. Mitchell, editors, *Machine Learning : An Artivicial Intelligence Approach.* Tioga Publiching Co, 1983.

[73] R. S. Michalski. Pattern recognition as rule-guided inference. *IEE Transactions on Pattern Analysis and Machine Intelligence*, 2(4):349–361, 1980.

[74] R. S. Michalski. *Machine Learning: An Artificial Intellligence Approach*, volume 1, chapter A theory and methodology of inductive learning. Morgan Kaufmann, Palo Alto, CA, 1983.

[75] T. M. Mitchell. Version spaces : An approach to concept learning. Technical report, Stanford Artificial Intelligence Laboratory, 1978.

[76] F. Montagna and G. Simi. Paradigms in measure theoretic learning and in informant learning. Unpublished Draft.

[77] R. J. Mooney and M. E. Califf. Induction of first-order decision lists : Results on learning the past tense of english verbs. *Journal ofArtificial Intelligence Research*, 3:1–24, 1995.

[78] R. J. Mooney and M. E. Califf. Learning the past tense of english verbs using inductive logic programming. In *Symbolic, Connectionist, and Statistical Approaches to Learning for Natural Language Processing*. Springer Verlag, 1996.

[79] K. Morik. Balanced cooperative modelling. In *Proc. First International Workshop on Multistrategy Learning*, pages 65–80, George Mason University, Fairfax, VA, 1991.

[80] K. Morik. Balanced cooperative modelling. *Machine Learning*, 11:217–235, 1993.

[81] K. Morik, S. Wrobel, J-U. Kietz, and W. Emde. *Knowledge Acquisition and Machine Learning: Theory, Methods and Applications.* Academic Press, 1993.

[82] S. Muggleton. Duce, an oracle based approach to constructive induction. In *Proceedings of the 10th International Joint Conference of Artificial Intelligence*, pages 287–292. Morgan Kaufmann, 1987.

[83] S. Muggleton. Inductive logic programming. *New Generation Computing*, 8:295–318, 1991.

[84] S. Muggleton, editor. *Inductive Logic Programming*. Academic Press, 1992.

[85] S. Muggleton. Inverting implication. In S. Muggleton and K. Furukawa, editors, *Proc. of the Second Int. Workshop on Inductive Logic Programming(ILP92)*, 1992.

[86] S. Muggleton. Predicate invention and utilisation. *Journal of Experimental and Theoretical Artificial Intelligence*, 6(1):127–130, 1994.

[87] S. Muggleton. Learning from positive data. In *Proceedings of the Inductive Logic Programming Workshop*, 1996.

[88] S. Muggleton and W. Buntine. Machine invention of first order predicates by inverting resolution. In R.S. Michalski, T.M. Mitchell, and J.G. Carbonell, editors, *Proceedings of the Fifth International Conference on Machine Learning*, pages 339–352. Morgan Kaufmann, 1988.

[89] S. Muggleton and L. De Raedt. Inductive Logic Programming: Theory and Methods. *Journal of Logic Programming*, 19(20):629–679, 1994.

[90] 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.

[91] S. Muggleton, R.D. King, and M.J.E. Sternberg. Protein secondary structure prediction using logic-based machine learning. *Protein Engineering*, 5(7):647–657, 1992.

[92] 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.

[93] S. Muggleton, A. Srinivasan, R. King, and M. Sternberg. Biochemical knowledge discovery using Inductive Logic Programming. In H. Motoda, editor, *Proc. of the first Conference on Discovery Science*, Berlin, 1998. Springer-Verlag.

[94] Stephen Muggleton. Inverse entailment and progol. *New Generation Computing Journal*, 13, May 1995.

[95] S.H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*. LNAI Tutorial 1228. Springer-Verlag, 1997.

[96] M. Pazzani and C. Brunk. Detecting and correcting errors in rule-based expert systems: an integration of empirical and explanation-based learning. *Knowledge Acquisition*, 3:157–173, 1991.

[97] M. Pazzani and D. Kibler. The utility of knowledge in inductive learning. *Machine Learning*, 9(1):57–94, 1992.

[98] G. Plotkin. A note on inductive generalisation. *Machine Intelligence*, 5:153–163, 1970.

[99] G. Plotkin. *Automatic methods of inductive inference*. PhD thesis, University of Edinburgh, 1971.

[100] G. Plotkin. A further note on inductive generalisation. *Machine Intelligence*, 6:101–124, 1971.

[101] K. Popper. *Conjectures and Refutations: the Growth of Scientific Knowledge*. Basic Books, New York, 1962.

[102] K. Popper. *The Logic of Scientific Discovery*. Harper Torch Books, New York, second edition, 1968.

[103] J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

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

[105] Ross Quinlan. Bagging, boosting, and c4.5. In *AAAI-96 Fourteenth National Conference on Artificial Intelligence, Portland*, 1996.

[106] Ross Quinlan. Boosting first-order learning. In *Machine Learning*, 1997.

[107] John C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intelligence*, 5:135–151, 1970.

[108] Elaine Rich. *Artificial Intelligence*. McGraw Hill, 1983.

[109] J. Rissanen. Modeling by shortest data description. *Auotomatica*, 14:465–471, 1978.

[110] J. A. Robinson. A machine-oriented logic based on the resolution princple. *Journal of the ACM*, 12:23–41, 1965.

[111] Sheldon M. Ross. *Introduction To Probability Models.* Academic Press Inc, 1989.

[112] D. E. Rumelhart and J. McClelland. On learning the past tense of English verbs. *Parallel Distributed Processing*, 2:216–271, 1986. MIT Press, Cambridge, MA.

[113] W. C. Salmon. *The foundtion of Scientific Inference.* University of Pittsburgh Press, 1966.

[114] C. Sammut. The origins of inductive logic programming : A prehistoric tale. In S. Muggleton, editor, *Third International Workshop on Inductive Logic Programming*, 1993.

[115] C.A. Sammut. *Learning Concepts by Performing Experiments.* PhD thesis, Department of Computer Science, University of New South Wales, 1981.

[116] C.A. Sammut and R.B. Banerji. Learning concepts by asking questions. In R. Michalski, J. Carbonell, and T. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach. Vol. 2*, pages 167–192. Morgan Kaufmann, 1986.

[117] R. E. Schapire. The strength of weak learnability. *Machine Learning*, 5(2):197–227, 1990.

[118] J. C. Schlimmer. *Concept acquisition through representational adjustment.* PhD thesis, Department of Information and Computer Science, University of California, Irvine, CA., 1987.

[119] G. Semeraro, F. Esposito, and D. Malerba. Learning contextual rules for document understanding. In *Proceedings of the tenth conference on Artifical Intelligence for applications*, pages 108–115, 1994.

[120] C. Shannon and W. Weaver. *The Mathematical Theory of Information.* University of Illinois Press, 1949.

[121] E. Shapiro. A general incremental algorithm that infers theory from facts. In *International Joint Conference on Artificial Intelligence*, 1981.

[122] E. Shapiro. Inductive inference of theories from facts. Technical Report 192, Computer Science Department, Yale University, 1981.

[123] A. Srinivasan, , R.D. King S.H. Muggleton, and M. Sternberg. Carcinogenesis predictions using ILP. In N. Lavrač and S. Džeroski, editors, *Proceedings of the Seventh International Workshop on Inductive Logic Programming*, pages 273–287. Springer-Verlag, Berlin, 1997.

[124] A. Srinivasan, S. Muggleton, R. King, and M. Sternberg. Mutagenesis: ILP experiments in a non-determinate biological domain. In S. Wrobel, editor, *Proceedings of the Fourth International Inductive Logic Programming Workshop.* Gesellschaft fur Mathematik und Datenverarbeitung MBH, 1994. GMD-Studien Nr 237.

[125] A. Srinivasan, S.H. Muggleton, R.D. King, and M.J.E. Sternberg. The predictive toxicology evaluation challenge. In *Fifteenth International Joint Conference on Artificial Intelligence*, volume 1, pages 4–9, 1997.

[126] I. Stahl. Compression measrures in ilp. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, pages 295–307, Amsterdam, 1996. IOS Press.

[127] S. Tangkitvanich and M. Shimura. Learning from an approximate theory and noisy examples. In *AAAI 93*, pages 466–471, 1993.

[128] S. A. Vere. Induction of concepts in the predicate calculus. In *Proceedings of the Fouth International Joint Conference on Artificial Intelligence*, pages 351–356, 1975.

[129] C.S. Wallace and D.M. Boulton. An information measure for classification. *Computing Journal*, 11:185–195, 1968.

[130] C.S. Wallace and P.R. Freeman. Estimation and inference by compact coding. *J. Royal Stat. Soc., Series B*, 49:240–251, 1987.

[131] R. E. Walpole and R. H. Myers. *Probability and Statistics for Engineers and Scientists.* Macmillan, fourth edition edition, 1989.

[132] Toby Walsh. Depth-boundid discrepancy search. In *Fifteenth International Joint Conference on Artificial Intelligence*, volume 2, pages 1388–1393, 1997.

[133] C. S. Wetherall. Probabilistic languages a review and some open questions. *ACM Computing Surveys*, 12(4):361–377, 1980.

[134] 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.

[135] X. Zhang and M. Numao. Learning and revising theories in noisy domains. In *ALT 97*, pages 339–351. Springer Verlag, 1997.

Appendix A

Statistics

A.1 Estimating the mean and standard error

The predictive accuracy is the key statistic for evaluating the performance of a learner. This tells how well the induced hypothesis will predict unseen examples. Generally, we only have a finite number of examples that cover a small portion of the instance spaces. These examples may be partitioned into training and test sets. Then the learner will induce a hypothesis using the training set. Then the test set will be used to assess the induced hypothesis. This gives an estimate of the predictive accuracy of this hypothesis on unseen examples.

If a sample consists of observations $x_1, x_2, \ldots x_n$ then the sample mean is:

$$\overline{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

The standard deviation δ is estimated by the sample standard deviation:

$$s = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \overline{x})^2}{n-1}}$$

The sample mean provides an estimation of the mean of the entire population. The more observations used in calculating the sample mean, the more accurately this will estimate the population mean. The central limit theory states that regardless of the population's distribution, the distribution of the sample mean will be approximately[1] normal, with mean μ and standard deviation $\frac{\delta}{\sqrt{n}}$. This standard deviation is known as the standard error.

We are interested in the accuracy of the sample mean. The standard error provides a measure of this accuracy. The standard error may be estimated by:

$$\frac{s}{\sqrt{n}}$$

[1]The approximation becomes better as n increases.

A.2 Comparing the average accuracy or error of two learners

This appendix is included to clarify the hypothesis testing used in the empirical studies when testing the significance of the difference between two means. This approach is used to compare two learning systems. If, for example, system A shows a lower average error than system B, then it would be inappropriate to conclude system A performs better than system B. For there is a chance these averages would result regardless of whether the average error of the two systems are identical.

To show, with an α level of confidence, that mean μ_1 is significantly greater then mean μ_2, then the null hypothesis is that the means are the same.

$$H_0 : \mu_1 = \mu_2$$

Now, the alternative hypothesis that is the mean of the first population is greater than that of the second.

$$H_1 : \mu_1 > \mu_2$$

Once, the sample data is collected, where the first set consists of n_1 readings and the second n_2, the sample means $\overline{x}_1$ and $\overline{x}_2$ may be calculated by the sample standard deviation s_1 and s_2. The value of the test statistic may now be calculated:

$$t = \frac{(\overline{x}_1 - \overline{x}_2)}{\sqrt{(s_1^2/n_1) + (s_2^2/n_2)}}$$

$$\nu = \frac{(s_1^2/n_1 + s_2^2/n_2)^2}{\frac{(s_1^2/n_1)^2}{n_1-1} + \frac{(s_2^2/n_2)^2}{n_2-1}}$$

The critical region is $t > t_\alpha$. Where ν is the degree of freedom. The value of t_α may be obtained from tables in any introductory book on statistics such as [131]. Therefore, if the value of the test statistic is in the critical region, we have an α level of significance in rejecting the null hypothesis.

When the identical tests are given to the two learning systems, that is both the training and test examples are identical, then the testing of the two means may be accomplished by pairing the results of the learning systems for each of the identical tests. The confidence interval for $\mu_1 - \mu_2$ in this case may use the test statistic:

$$t = \frac{\overline{d} - d_0}{s_d/\sqrt{n}}$$

The critical regions may be constricted with the *t-distribution* with $n - 1$ degrees of freedom.

Appendix B

Lemmas for stochastic limit results

Lemma B.0.1 *Suppose* $0 \leq \epsilon < 1$, $0 \leq \mu_p \leq \epsilon$, $0 < \theta_t \leq 1$, $0 < \delta_a + \delta_b$, $0 \leq \delta_a \leq \theta_t$, *and* $0 \leq \delta_b \leq 1 - \theta_t$. *then* $e_p(\epsilon, \mu_p, \theta_t, \delta_a, \delta_b) > 0$ *where*

$$\begin{aligned} e_p(\epsilon, \mu_p, \theta_t, \delta_a, \delta_b) =& (1 - \mu_p + \mu_p\theta_t)\lg(\frac{1-\epsilon}{\theta_t} + \epsilon) \\ & - ((1-\mu_p)(1 - \frac{\delta_a}{\theta_t}) + \mu_p(\theta_t - \delta_a) + \mu_p\delta_b))\lg(\frac{1-\epsilon}{\theta_t - \delta_a + \delta_b} + \epsilon) \\ & - ((1-\mu_p)\frac{\delta_a}{\theta_t} + \mu_p(\delta_a - \delta_b))\lg(\epsilon) \end{aligned}$$

PROOF:

Consider the case when $\epsilon = \mu_p = 0$ then $e_p = \lg(\frac{1}{\theta_t}) - (1 - \frac{\delta_a}{\theta_t})\lg(\frac{1}{\theta_t - \delta_a + \delta_b}) - (\frac{\delta_a}{\theta_t})(-\infty) > 0$ as if $\delta_a > 0$ then $e_p = \infty$ and if $\delta_a = 0$ then $e_p \geq \lg(\frac{1}{\theta_t}) - \lg(\frac{1}{\theta_t + \delta_b}) > 0$. Now, assuming $\epsilon \neq 0$, e_p may be simplified.

$$\begin{aligned} e_p(\epsilon, \mu_p, \theta_t, \delta_a, \delta_b) =& (1 - \mu_p + \mu_p\theta_t)\lg(\frac{\frac{1}{\epsilon} - 1}{\theta_t} + 1) - \\ & (1 - \mu_p + \mu_p\theta_t - \delta_a(\frac{1-\mu_p}{\theta_t} + \mu_p) + \delta_b\mu_p)\lg(\frac{\frac{1}{\epsilon} - 1}{\theta_t - \delta_a + \delta_b} + 1) \end{aligned}$$

To show $e_p > 0$ involves the following steps:

- Step 1 we show $e_p > 0$ for $\mu_p = 0$.
- Step 2 we show $e_p > 0$ for $\mu_p = \epsilon$.
- Step 3 we show that e_p is affine with respect to μ_p.

Hence, $e_p > 0$ for $0 \leq \mu_p \leq \epsilon$, as, it is greater than zero at the end points and is a straight line within the region in question.

Step 1: We wish to show $e_p|_{\mu_p = 0} > 0$ for $0 < \delta_a + \delta_b$, $0 < \delta_a \leq \theta_t$, and $0 < \delta_b \leq 1 - \theta_t$. This proceed by dividing the proof up into 5 cases. These cases correspond to regions on an argand

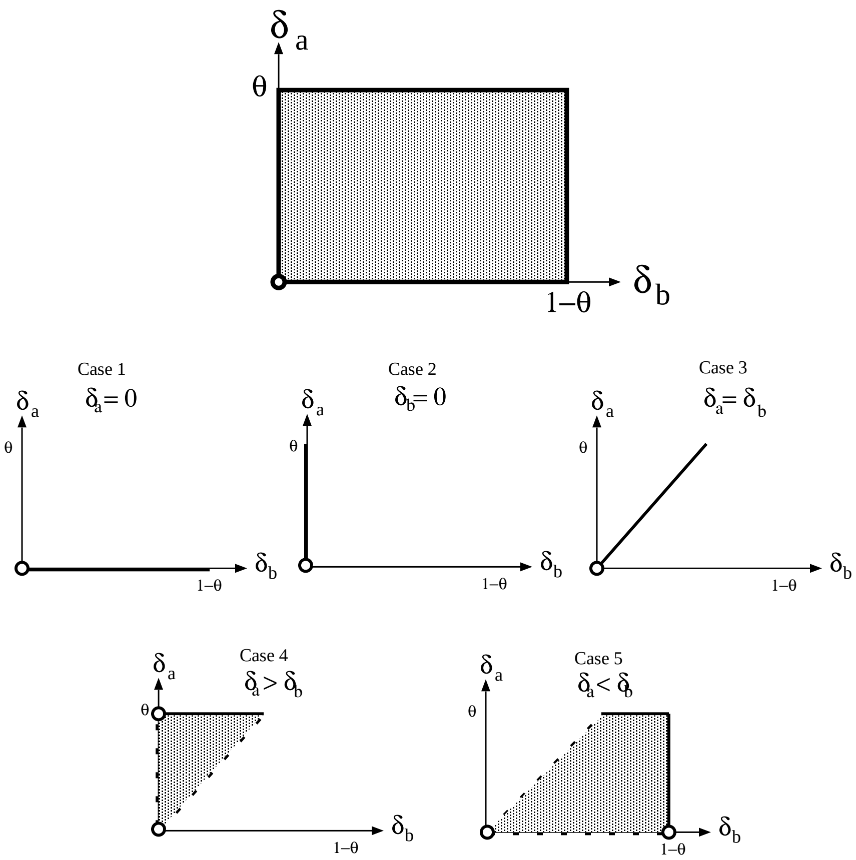


Figure B.1: Division of region into 5 separate cases.

plane, with δ_a for the vertical dimension and δ_b for the horizontal. This is shown in Figure B.1. By applying the substitution $\mu_p = 0$, e_p simplifies to the following:

$$e_p|_{\mu_p=0} = \lg(\frac{\frac{1}{\epsilon}-1}{\theta_t}+1) - (1-\frac{\delta_a}{\theta_t})\lg(\frac{\frac{1}{\epsilon}-1}{\theta_t-\delta_a+\delta_b}+1)$$

Case 1. $\delta_a = 0$ Since $\delta_b > 0$, $\frac{\frac{1}{\epsilon}-1}{\theta_t}+1 > \frac{\frac{1}{\epsilon}-1}{\theta_t+\delta_b}+1$. Hence:

$$e_p|_{\substack{\mu_p=0\\ \delta_a=0}} = \lg(\frac{\frac{1}{\epsilon}-1}{\theta_t}+1) - \lg(\frac{\frac{1}{\epsilon}-1}{\theta_t+\delta_b}+1) > 0$$

Case 2. $\delta_b = 0$

$$e_p|_{\substack{\mu_p=0\\ \delta_b=0}} = \lg(\frac{\frac{1}{\epsilon}-1}{\theta_t}+1) - (1-\frac{\delta_a}{\theta_t})\lg(\frac{\frac{1}{\epsilon}-1}{\theta_t-\delta_a}+1)$$

The partial derivative with respect to δ_a is:

$$\frac{\partial e_p|_{\substack{\mu_p=0\\ \delta_b=0}}}{\partial\delta_a} = \frac{1}{\theta_t\ln(2)}(\ln(\frac{\frac{1}{\epsilon}-1}{\theta_t-\delta_a}+1) - \frac{\frac{1}{\epsilon}-1}{\frac{1}{\epsilon}-1+\theta_t-\delta_a})$$

Let $x = \theta_t - \delta_a$ and to simplify the notation let $f_1(\epsilon, x) = \frac{\partial e_p|_{\mu_p=0, \delta_b=0}}{\partial \delta_a}$. Due to the restrictions on δ_a and θ_t, x will be restricted to $0 \le x < 1$. Now,

$$f_1(\epsilon, x) = \frac{1}{\theta_t \ln(2)}(\ln(\frac{\frac{1}{\epsilon} - 1 + x}{x}) - \frac{\frac{1}{\epsilon} - 1}{\frac{1}{\epsilon} - 1 + x})$$

$$\frac{\partial f_1}{\partial x} = \frac{-1}{\theta_t \ln(2)} \frac{1 - \frac{2}{\epsilon} + (\frac{1}{\epsilon})^2}{x(\frac{1}{\epsilon} - 1 + x)^2} < 0$$

As f_1 is positive at the end points, $\lim_{x \to 0} f_1(\epsilon, x) = \infty$ and $f_1(\epsilon, 1) = \frac{1}{\theta_t \ln(2)}(\epsilon - \ln(e) - 1) > 0$, and the gradient is always negative between these points without any singularities, the function f_1 is positive for $0 \le x < 1$. Hence, $e_p|_{\mu_p=0, \delta_b=0}$ increases for $\delta_a > 0$. And as $e_p|_{\mu_p=0, \delta_a=0, \delta_b=0} = 0$ and the function is positive for $\delta_a > 0$.

Case 3. $\delta_a = \delta_b$

$$e_p\big|_{\substack{\mu_p=0 \\ \delta_a=\delta_b}} = \frac{\delta_a}{\theta_t} \lg(\frac{\frac{1}{\epsilon} - 1}{\theta_t} + 1) > 0 \text{ for } \delta_a > 0$$

Case 4. $\delta_a > \delta_b$

Let $\delta_x = \delta_a - \delta_b$.

$$\begin{aligned} e_p|_{\mu_p=0} &= \lg(\frac{\frac{1}{\epsilon} - 1}{\theta_t} + 1) - (1 - \frac{\delta_x + \delta_b}{\theta_t}) \lg(\frac{\frac{1}{\epsilon} - 1}{\theta_t - \delta_x} + 1) \\ &> \lg(\frac{\frac{1}{\epsilon} - 1}{\theta_t} + 1) - (1 - \frac{\delta_x}{\theta_t}) \lg(\frac{\frac{1}{\epsilon} - 1}{\theta_t - \delta_x} + 1) \\ &> 0 \quad \text{(as shown in Case 2)} \end{aligned}$$

Case 5. $\delta_a < \delta_b$

Let $\delta_x = \delta_b - \delta_a$.

$$\begin{aligned} e_p|_{\mu_p=0} &= \lg(\frac{\frac{1}{\epsilon} - 1}{\theta_t} + 1) - (1 - \frac{\delta_a}{\theta_t}) \lg(\frac{\frac{1}{\epsilon} - 1}{\theta_t + \delta_x} + 1) \\ &> \lg(\frac{\frac{1}{\epsilon} - 1}{\theta_t} + 1) - (1 - \frac{\delta_a}{\theta_t}) \lg(\frac{\frac{1}{\epsilon} - 1}{\theta_t} + 1) \\ &= \frac{\delta_a}{\theta_t} \lg(\frac{\frac{1}{\epsilon} - 1}{\theta_t} + 1) \\ &> 0 \end{aligned}$$

Step 2: The proof proceeds by showing that $e_p > 0$ when $\mu_p = \epsilon$, for any $\epsilon \in [0, 1)$. This in conjunction with the previous step shows that $e_p > 0$ over all considered values of μ_p, ϵ, δ_a, δ_b, for all of the cases of interest.

The proof is broken up into 5 cases. These are the same regions previously used when considering $\mu_p = 0$ and are shown in Figure B.1.

Case 1. $\delta_a = 0$ Since $e_p|_{\delta_a=\delta_b=0} = 0$ and the gradient of $e_p|_{\delta_a=0}$ is positive for $\epsilon = \mu_p$, $e_p|_{\delta_a=0}$ is also positive for $\delta_b > 0$. Hence:

$$e_p|_{\delta_a=0} = (1-\mu_p+\mu_p\theta_t)\lg(\frac{\frac{1}{\epsilon}-1}{\theta_t}+1) - (1-\mu_p+\mu_p\theta_t+\mu_p\delta_b)\lg(\frac{\frac{1}{\epsilon}-1}{\theta_t+\delta_b}+1)$$

$$\left.\frac{\partial e_p|_{\delta_a=0}}{\partial \delta_b}\right|_{\epsilon=\mu_p} = \frac{\mu_p}{\ln(2)}\left(\frac{\frac{1}{\mu_p}-1}{\theta_t+\delta_b} - \ln(\frac{\frac{1}{\mu_p}-1}{\theta_t+\delta_b}+1)\right) > 0$$

Note, that $\frac{\frac{1}{\mu_p}-1}{\theta_t+\delta_b} > 0$ and $x - \ln(x+1) > 0$ for $x > 0$.
Case 2. $\delta_b = 0$

$$e_p|_{\delta_b=0} = (1-\mu_p+\mu_p\theta_t)\lg(\frac{\frac{1}{\epsilon}-1}{\theta_t}+1) - (1-\mu_p+\mu_p\theta_t-\delta_a(\frac{1-\mu_p}{\theta_t}+\mu_p))\lg(\frac{\frac{1}{\epsilon}-1}{\theta_t-\delta_a}+1)$$

To show $e_p|_{\delta_b=0} > 0$ for $\mu_p = \epsilon$ it suffices to prove that $e_p|_{\delta_b=0} > 0$ for any $0 < \epsilon < 1$ and $\mu_p = 1$.

$$e_p|_{\substack{\delta_b=0\\ \mu_p=1}} = \theta_t\lg(\frac{\frac{1}{\epsilon}-1}{\theta_t}+1) - (\theta_t-\delta_a)\lg(\frac{\frac{1}{\epsilon}-1}{\theta_t-\delta_a}+1)$$

$$\frac{\partial e_p|_{\substack{\delta_b=0\\ \mu_p=1}}}{\partial \delta_a} = \frac{1}{\ln(2)}\left(\ln(\frac{\frac{1}{\epsilon}-1+\theta_t-\delta_a}{\theta_t-\delta_a}) - (\frac{\frac{1}{\epsilon}-1}{\frac{1}{\epsilon}-1+\theta_t-\delta_a})\right)$$

Let $x = \theta_t - \delta_a$. Given the previous restrictions on θ_t and δ_a, x will be restricted to $[0, 1)$. Also, let $f_1 = \frac{\partial e_p|_{\substack{\delta_b=0\\ \mu_p=1}}}{\partial \delta_a}$ to simplify the notation. So

$$f_1(\epsilon, x) = \frac{1}{\ln(2)}(\ln(\frac{\frac{1}{\epsilon}-1+x}{x}) - (\frac{\frac{1}{\epsilon}-1}{\frac{1}{\epsilon}-1+x}))$$

It is clear that $\lim_{x\to 0} f_1 = \infty$ and $f_1|_{x=1} = \frac{1}{\ln(2)}(\epsilon - 1 - \ln(\epsilon)) > 0$. Also,

$$\frac{\partial f_1}{\partial x} = \frac{-1}{\ln(2)}\left(\frac{1-\frac{2}{\epsilon}+(\frac{1}{\epsilon})^2}{(\frac{1}{\epsilon}-1+x)^2 x}\right) < 0$$

So $f_1 > 0$ for $0 \le x \le 1$, as the end points are positive and the function is decreasing across this range without any singularities. Hence, $e_p|_{\delta_b=0} > 0$ as $e_p|_{\substack{\delta_a=0\\ \delta_b=0}} = 0$ and the function is increasing for $\delta_a \ge 0$ for any $0 < \epsilon < 1$ and $\mu_p = 1$. This in turn implies $e_p|_{\delta_b=0} > 0$ for $\mu_p = \epsilon$ as e_p is affine in μ_p and $e_p|_{\delta_b=0} > 0$ for $\mu_p = 0$.
Case 3. $\delta_a = \delta_b$

$$e_p|_{\delta_a=\delta_b} = \delta_a(\frac{1-\mu_p}{\theta_t})\lg(\frac{\frac{1}{\epsilon}-1}{\theta_t}+1) > 0$$

Case 4. $\delta_a > \delta_b$

Let $\delta_x = \delta_a - \delta_b$.

$$\begin{aligned}
e_p =&(1-\mu_p+\mu_p\theta_t)\lg(\frac{\frac{1}{\epsilon}-1}{\theta_t}+1) - \\
&(1-\mu_p+\mu_p\theta_t-\delta_x(\frac{1-\mu_p}{\theta_t}+\mu_p)-\delta_b(\frac{1-\mu_p}{\theta_t}))\lg(\frac{\frac{1}{\epsilon}-1}{\theta_t-\delta_x}+1) \\
>&(1-\mu_p+\mu_p\theta_t)\lg(\frac{\frac{1}{\epsilon}-1}{\theta_t}+1)-(1-\mu_p+\mu_p\theta_t-\delta_x(\frac{1-\mu_p}{\theta_t}+\mu_p))\lg(\frac{\frac{1}{\epsilon}-1}{\theta_t-\delta_x}+1) \\
>&0 \quad \text{(as shown in case 2)}
\end{aligned}$$

Case 5. $\delta_a < \delta_b$

Let $\delta_x = \delta_b - \delta_a$.

$$\begin{aligned}
e_p &= (1-\mu_p+\mu_p\theta_t)\lg(\frac{\frac{1}{\epsilon}-1}{\theta_t}+1)-(1-\mu_p+\mu_p\theta_t-\delta_a(\frac{1-\mu_p}{\theta_t})-\delta_x\mu_p)\lg(\frac{\frac{1}{\epsilon}-1}{\theta_t+\delta_x}+1) \\
&> (1-\mu_p+\mu_p\theta_t)\lg(\frac{\frac{1}{\epsilon}-1}{\theta_t}+1)-(1-\mu_p+\mu_p\theta_t+\delta_x\mu_p)\lg(\frac{\frac{1}{\epsilon}-1}{\theta_t+\delta_x}+1) \\
&> 0 \quad \text{(as shown in case 1)}
\end{aligned}$$

Step 3: e_p is affine, that is a straight line, with respect to μ_p because $\frac{\partial e_p}{\partial \mu_p}$ is a constant. This is seen when the derivative of e_p with respect to μ_p is calculated.

$$\frac{\partial e_p}{\partial \mu_p} = (1-\theta_t-\frac{\delta_a}{\theta_t}+\delta_a-\delta_b)\lg(\frac{\frac{1}{\epsilon}-1}{\theta_t-\delta_a+\delta_b}+1)-(1-\theta_t)\lg(\frac{\frac{1}{\epsilon}-1}{\theta_t}+1)$$

By combining these three steps it is clear that $e_p > 0$ for any $0 \le \mu_p \le \epsilon$, as e_p is greater than 0 at the end points and a straight line within the boundary. This completes the proof of this lemma. ■

Lemma B.0.2 *Suppose* $0 \le \epsilon < 1$, $0 \le \mu_p \le \epsilon$, $0 < \theta_t \le 1$, $0 \le \delta_a \le \theta_t$, *and* $0 \le \delta_b \le 1-\theta_t$. *Then there exists constants* $c_1, c_2 \in \mathbb{R}$ *such that for all* $\delta \le c_1$, $e_p(\epsilon,\mu_p,\theta_t,\delta_a,\delta_b) \ge c_2\delta$ *where* $\delta = \delta_a + \delta_b$, $c_1 > 0$, $c_2 > 0$, *and*

$$\begin{aligned}
e_p(\epsilon,\mu_p,\theta_t,\delta_a,\delta_b) =&(1-\mu_p+\mu_p\theta_t)\lg(\frac{1-\epsilon}{\theta_t}+\epsilon) \\
&-((1-\mu_p)(1-\frac{\delta_a}{\theta_t})+\mu_p(\theta_t-\delta_a)+\mu_p\delta_b))\lg(\frac{1-\epsilon}{\theta_t-\delta_a+\delta_b}+\epsilon) \\
&-((1-\mu_p)\frac{\delta_a}{\theta_t}+\mu_p(\delta_a-\delta_b))\lg(\epsilon)
\end{aligned}$$

PROOF:

Essentially this proof involves showing that there is a "wedge" that e_p always is at or above. This is diagrammatically shown in Figure B.2.

Consider the case when $\epsilon = \mu_p = 0$. Then $e_p = \lg(\frac{1}{\theta_t}) - (1-\frac{\delta_a}{\theta_t})\lg(\frac{1}{\theta_t-\delta_a+\delta_b}) - (\frac{\delta_a}{\theta_t})(-\infty) > 0$. Let $c_1 = 1-\theta_t$ and $c_2 = \frac{\lg(\frac{1}{\theta_t})}{1-\theta_t}$. So the case of interest is when $0 \le \delta \le 1-\theta_t$. First, consider the case when $\delta_a > 0$; then $e_p = \infty \ge c_2\delta$ for all $\delta \le c_1$. Second, consider when $\delta_a = 0$ in which case $\delta_b = \delta$ and and let $f(\theta_t,\delta_b) = \lg(\frac{1}{\theta_t}) - \lg(\frac{1}{\theta_t+\delta_b})$; then $e_p \ge f(\theta_t,\delta_b) \ge c_2\delta$ for all $\delta \le c_1$. $f(\theta_t,\delta)$ is equal to $c_2\delta$ at the end points of this range of interest, $f(\theta_t,0)=0$, and $f(\theta_t,1-\theta_t) = \frac{\lg(\frac{1}{\theta_t})}{1-\theta_t}(1-\theta_t)$.

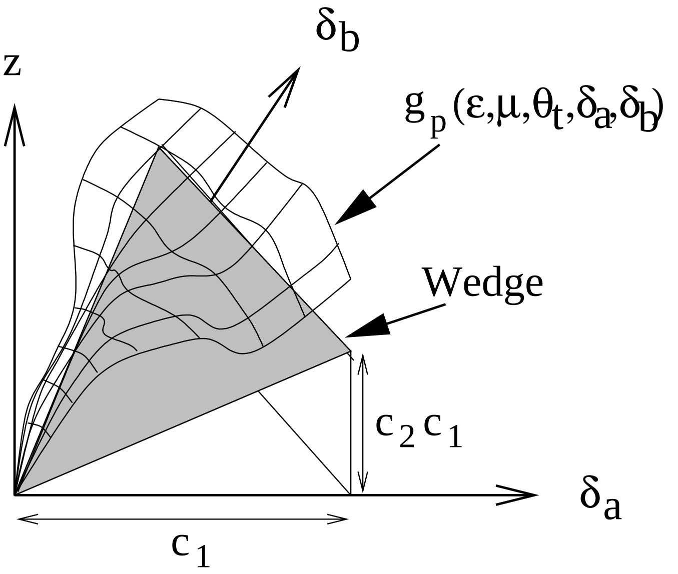


Figure B.2: e_p is shown to be at or above the "wedge".

Also, $f(\theta_t, \delta)$ is continuous, $\frac{\partial f}{\partial \delta} > 0$, and concave down, over this range. $\frac{\partial f}{\partial \delta_b} = \frac{1}{(\theta_t+\delta_b)\ln(2)}$, and $\frac{\partial^2 f}{\partial {\delta_b}^2} = -\frac{1}{(\theta_t+\delta_b)^2 \ln(2)}$. Note, if $\theta_t = 1$ then $\delta_b = 0$.

Now, assuming $\epsilon \neq 0$, e_p may be simplified.

$$\begin{aligned} e_p(\epsilon, \mu_p, \theta_t, \delta_a, \delta_b) =& (1 - \mu_p + \mu_p\theta_t) \lg(\frac{\frac{1}{\epsilon} - 1}{\theta_t} + 1) - \\ & (1 - \mu_p + \mu_p\theta_t - \delta_a(\frac{1-\mu_p}{\theta_t} + \mu_p) + \delta_b\mu_p) \lg(\frac{\frac{1}{\epsilon} - 1}{\theta_t - \delta_a + \delta_b} + 1) \end{aligned}$$

The proof is now divided into three steps:

- *Step 1* We show that there exists a constant $c_3 > 0$ such that the minimum of e_p, for fixed variables $\delta \leq c_3$, ϵ, μ_p, and θ_t , is reached either when $\delta_a = 0$ and $\delta_b = \delta$ or when $\delta_a = \delta$ and $\delta_b = 0$, for $0 \leq \delta_b \leq \delta$ and $\delta_a = \delta - \delta_b$. This is achieved by partitioning the proof into two cases: $\mu_p < \epsilon$, and $\mu_p = \epsilon$. The first is shown by fixing δ for some $\delta < c_3$ and proving that e_p is concave down. The second is similarly shown by fixing δ for some $\delta < c_3$ but in this case e_p gradient is shown to be always negative.

- *Step 2* We evaluate the gradient of e_p with respect to δ_a when $\delta_b = 0$ at $\delta_a = 0$ and we show this to be positive in all cases of interest. Correspondingly, we evaluate the gradient of e_p with respect to δ_b when $\delta_a = 0$ at $\delta_b = 0$ and we show this to be positive also.

- *Step 3* We show, using the fundamental definition of differentiation and limits, that there exists a "wedge" that is at or below e_p when either $\delta_a = 0$ or $\delta_b = 0$. Hence, this "wedge" will be lower then e_p as one of these boarder cases will be the minimum of e_p for a fixed δ.

This completes the proof, showing the required result.

Step 1 This step is divided into two cases. The first case is when $\mu_p < \epsilon$ and the second when $\mu_p = \epsilon$.

Assume $\mu_p < \epsilon$. Suppose δ is fixed and let $\delta_a = \delta - \delta_b$. Then $0 \leq \delta_b \leq \delta$ and e_p may be rewritten with this substitution.

$$\begin{aligned} e_p(\epsilon, \mu_p, \theta_t, \delta, \delta_b) =& (1 - \mu_p + \mu_p\theta_t) \lg(\frac{\frac{1}{\epsilon} - 1}{\theta_t} + 1) - \\ & (1 - \mu_p + \mu_p\theta_t - (\delta - \delta_b)(\frac{1 - \mu_p}{\theta_t} + \mu_p) + \delta_b\mu_p) \lg(\frac{\frac{1}{\epsilon} - 1}{\theta_t - \delta + 2\delta_b} + 1) \end{aligned}$$

The second derivative of e_p with respect to δ_b many be calculated and simplified.

$$\frac{\partial^2 e_p(\epsilon, \mu_p, \theta_t, \delta, \delta_b)}{\partial \delta_b^2} = \frac{4(\epsilon - 1)(\theta_t^2(\epsilon - \mu_p) + \delta_b w_1 + \delta w_2 + \delta\delta_b w_3 + \delta^2 w_4)}{\ln(2)\theta_t(\theta_t - \delta + 2\delta_b)^2(1 - \epsilon + \epsilon\theta_t - \epsilon\delta + 2\epsilon\delta_b)^2}$$

where $w_1 = \mu_p - 1 + \epsilon - \mu_p\epsilon + 2\theta_t\epsilon - 2\mu_p\theta_t$, $w_2 = \mu_p\theta_t + \mu_p\theta_t\epsilon - 2\theta_t\epsilon$, $w_3 = 2\mu_p\epsilon - 2\epsilon$, and $w_4 = \epsilon - \mu_p\epsilon$. Let $c_3 = \frac{1}{2}\frac{\theta_t^2(\epsilon - \mu_p)}{|w_1| + |w_2| + |w_3| + |w_4|}$. Consider $\delta \leq c_3$.

$$\begin{aligned} & \delta \leq \frac{1}{2}\frac{\theta_t^2(\epsilon - \mu_p)}{|w_1| + |w_2| + |w_3| + |w_4|} \\ \Rightarrow & \delta < \frac{\theta_t^2(\epsilon - \mu_p)}{|w_1| + |w_2| + |w_3| + |w_4|} \\ \Rightarrow & \delta|w_1| + \delta|w_2| + \delta|w_3| + \delta|w_4| < \theta_t^2(\epsilon - \mu_p) \\ \Rightarrow & \delta_b|w_1| + \delta|w_2| + \delta\delta_b|w_3| + \delta^2|w_4| < \theta_t^2(\epsilon - \mu_p) \\ \Rightarrow & 0 < \theta_t^2(\epsilon - \mu_p) - \delta_b|w_1| - \delta|w_2| - \delta\delta_b|w_3| - \delta^2|w_4| \\ \Rightarrow & 0 < \theta_t^2(\epsilon - \mu_p) + \delta_b w_1 + \delta w_2 + \delta\delta_b w_3 + \delta^2 w_4 \end{aligned}$$

Assuming $\mu_p < \epsilon$ and $\delta < c_3$, e_p will be concave down, that is $\frac{\partial^2 e_p(\epsilon, \mu_p, \theta_t, \delta, \delta_b)}{\partial \delta_b^2} < 0$, for all $0 \leq \delta_b \leq \delta$. Hence, the minimum of e_p for a $\delta \leq c_3$ will be either at $\delta_a = \delta$ and $\delta_b = 0$ or $\delta_a = 0$ and $\delta_b = \delta$.

Now consider the case when $\mu_p = \epsilon$. Substitute $\epsilon = \mu_p$ and $\delta_a = \delta - \delta_b$ in e_p. This yields:

$$\begin{aligned} e_p(\mu_p, \theta_t, \delta, \delta_b) =& (1 - \mu_p + \mu_p\theta_t) \lg(\frac{\frac{1}{\mu_p} - 1}{\theta_t} + 1) - \\ & (1 - \mu_p + \mu_p\theta_t - (\delta - \delta_b)(\frac{1 - \mu_p}{\theta_t} + \mu_p) + \delta_b\mu_p) \lg(\frac{\frac{1}{\mu_p} - 1}{\theta_t - \delta + 2\delta_b} + 1) \end{aligned}$$

The partial derivative with respect to δ_b may be calculated.

$$\begin{aligned} \frac{\partial e_p(\mu_p, \theta_t, \delta, \delta_b)}{\partial \delta_b} = \frac{1}{\ln(2)}\Bigg(& \frac{2(1 - \mu_p + \mu_p\theta_t - (\delta - \delta_b)(\frac{1-\mu_p}{\theta_t} + \mu_p) + \delta_b\mu_p)(\frac{1}{\mu_p} - 1)}{(\theta_t - \delta + 2\delta_b)^2(\frac{\frac{1}{\mu_p} - 1}{\theta_t - \delta + 2\delta_b} + 1)} - \\ & (\frac{1 - \mu_p}{\theta_t} + 2\mu_p) \ln(\frac{\frac{1}{\mu_p} - 1}{\theta_t - \delta + 2\delta_b} + 1)\Bigg) \end{aligned}$$

We now consider the limit as δ goes to zero. Note, δ_b is bound by δ, that is $0 \leq \delta_b \leq \delta$. Let $x = \frac{\mu_p\theta_t}{1 - \mu_p + \mu_p\theta_t}$. Then:

$$\lim_{\delta\to 0}\frac{\partial e_p(\mu_p,\theta_t,\delta,\delta_b)}{\partial \delta_b} = \frac{1}{\theta_t\ln(2)}\Big((1-\mu_p+2\mu_p\theta_t)\ln(\frac{\mu_p\theta_t}{1-\mu_p+\mu_p\theta_t})+2(1-\mu_p)\Big)$$
$$= \frac{\mu_p}{x(1-\mu_p)\ln(2)}\Big((1+x-\mu_p-\mu_p x)\ln(x)+2(1-\mu_p)(1-x)\Big)$$

Set $f(\mu_p,x)=(1+x-\mu_p-\mu_p x)\ln(x)+2(1-\mu_p)(1-x)$ for all $x\in[0,\mu_p)$. Then $f(\mu_p,x)|_{\mu_p=1}=0$ and $\frac{\partial f(\mu_p,x)}{\partial \mu_p} = -2(1-x)-(1+x)\ln(x)\geq 0$ for $0<x<1$. Hence $f(\mu_p,x)<0$ for the cases of interest. Moreover, $\lim_{\delta\to 0}\frac{\partial e_p(\mu_p,\theta_t,\delta,\delta_b)}{\partial\delta_b}<0$ for $0<\mu_p<1$, and $0<\theta_t\leq 1$. Let $g(x)=-2(1-x)-(1+x)\ln(x)$. Then $g(x)\geq 0$ for $0<x<1$ as $g(1)=0$ and $\frac{dg}{dx}=2-\ln(x)-\frac{1+x}{x}$, $\frac{d^2g}{dx^2}=\frac{1-x}{x^2}$, so $\frac{dg}{dx}|_{x=1}=0$ and g is concave up for $0<x<1$.

Let $\xi=\lim_{\delta\to 0}\frac{\partial e_p(\mu_p,\theta_t,\delta,\delta_b)}{\partial\delta_b}$. Then, as e_p is continuous in δ and δ_b, there is a $c_3>0$ such that for all $\delta<c_3$, $\frac{\partial e_p(\mu_p,\theta_t,\delta,\delta_b)}{\partial\delta_b}<\frac{1}{2}\xi$. So for $\delta<c_3$, e_p will be minimum when $\delta_b=\delta$.

Step 2 We assume ϵ, μ_p, and θ_t are constants with the following restrictions: $0<\epsilon<1$, $0\leq\mu_p\leq\epsilon$, and $0<\theta_t\leq 1$ respectively. So, we are considering e_p with respect to the two variables δ_a and δ_b. Note, e_p is continuous with respect to δ_a and δ_b at $\delta_a=\delta_b=0$. We now calculate the partial derivative with respect to δ_a and then with respect to δ_b.

$$\frac{\partial e_p(\epsilon,\mu_p,\theta_t,\delta_a,\delta_b)}{\partial\delta_a} = \frac{1}{\ln(2)}\Bigg(\Big(\frac{1-\mu_p}{\theta_t}+\mu_p\Big)\ln\Big(\frac{\frac{1}{\epsilon}-1}{\theta_t-\delta_a+\delta_b}+1\Big)- \frac{(1-\mu_p+\mu_p\theta_t-\delta_a(\frac{1-\mu_p}{\theta_t}+\mu_p)+\mu_p\delta_b)(\frac{1}{\epsilon}-1)}{(\theta_t-\delta_a+\delta_b)^2(\frac{\frac{1}{\epsilon}-1}{\theta_t-\delta_a+\delta_b}+1)}\Bigg)$$

$$\frac{\partial e_p(\epsilon,\mu_p,\theta_t,\delta_a,\delta_b)}{\partial\delta_b} = \frac{1}{\ln(2)}\Bigg(-\mu_p\ln\Big(\frac{\frac{1}{\epsilon}-1}{\theta_t-\delta_a+\delta_b}+1\Big)+ \frac{(1-\mu_p+\mu_p\theta_t-\delta_a(\frac{1-\mu_p}{\theta_t}+\mu_p)+\mu_p\delta_b)(\frac{1}{\epsilon}-1)}{(\theta_t-\delta_a+\delta_b)^2(\frac{\frac{1}{\epsilon}-1}{\theta_t-\delta_a+\delta_b}+1)}\Bigg)$$

Note, both partial derivatives are continuous at the origin. First, we consider $\frac{\partial e_p}{\partial\delta_a}|_{\delta_a=\delta_b=0}$.

$$\frac{\partial e_p(\epsilon,\mu_p,\theta_t,\delta_a,\delta_b)}{\partial\delta_a}|_{\delta_a=\delta_b=0} = \frac{1}{\ln(2)}\Bigg(\Big(\frac{1-\mu_p}{\theta_t}+\mu_p\Big)\ln\Big(\frac{\frac{1}{\epsilon}-1}{\theta_t}+1\Big)- \frac{(1-\mu_p+\mu_p\theta_t)(\frac{1}{\epsilon}-1)}{\theta_t^2(\frac{\frac{1}{\epsilon}-1}{\theta_t}+1)}\Bigg)$$

We show this to be positive over all possible values of ϵ, μ_p, and θ_t. Note, this function is affine with respect to μ_p, hence it suffices to show it to be positive for $\mu_p=0$ and $\mu_p=\epsilon$: clearly, it will be positive for all values in-between $0\leq\mu_p\leq\epsilon$. Consider $\mu_p=0$. Let $x=\frac{1}{\epsilon}-1$ (hence

$x > 0$).

$$\frac{\partial e_p(\epsilon,\mu_p,\theta_t,\delta_a,\delta_b)}{\partial \delta_a}|_{\substack{\delta_a=\delta_b=0\\ \mu_p=0}} = \frac{1}{\ln(2)}\left(\left(\frac{1}{\theta_t}\right)\ln\left(\frac{\frac{1}{\epsilon}-1}{\theta_t}+1\right) - \frac{(\frac{1}{\epsilon}-1)}{\theta_t^2(\frac{\frac{1}{\epsilon}-1}{\theta_t}+1)}\right)$$
$$= \frac{1}{\ln(2)\theta_t}\left(\ln(\frac{x}{\theta_t}+1) - \frac{x}{x+\theta_t}\right)$$
$$= f_1(\theta_t,x) \qquad \text{(this simplifies the notation)}$$

Note $f_1(\theta_t,x)|_{x=0} = 0$ and $\frac{\partial f_1(\theta_t,x)}{\partial x} = \frac{x}{\ln(2)\theta_t(x+\theta_t)^2}$ which is positive for all $x > 0$. Hence $f_1(\theta_t,x)$ is positive for all $x > 0$.

Now consider the other end point $\mu_p = \epsilon$. Let $x = \frac{\frac{1}{\epsilon}-1}{\theta_t}$ (hence $x > 0$). To simplify the notation, let $f_2(x) = (x+1)\ln(x+1) - x$.

$$\frac{\partial e_p(\epsilon,\mu_p,\theta_t,\delta_a,\delta_b)}{\partial \delta_a}|_{\substack{\delta_a=\delta_b=0\\ \mu_p=\epsilon}} = \frac{1}{\ln(2)}\left(\left(\frac{1-\epsilon}{\theta_t}+\epsilon\right)\ln\left(\frac{\frac{1}{\epsilon}-1}{\theta_t}+1\right) - \frac{(1-\epsilon+\epsilon\theta_t)(\frac{1}{\epsilon}-1)}{\theta_t^2(\frac{\frac{1}{\epsilon}-1}{\theta_t}+1)}\right)$$
$$= \frac{\epsilon}{\ln(2)}((x+1)\ln(x+1) - x)$$
$$= \frac{\epsilon}{\ln(2)} f_2(x)$$

Now $f_2(x)|_{x=0} = 0$ and $\frac{df_2}{dx} = \ln(x+1)$ which is positive for all $x > 0$. Hence $\frac{\partial e_p(\epsilon,\mu_p,\theta_t,\delta_a,\delta_b)}{\partial \delta_a}|_{\substack{\delta_a=\delta_b=0\\ \mu_p=\epsilon}}$ is also positive over the desired range of θ_t and ϵ.

Second, we consider $\frac{\partial e_p}{\partial \delta_b}|_{\delta_a=\delta_b=0}$.

$$\frac{\partial e_p(\epsilon,\mu_p,\theta_t,\delta_a,\delta_b)}{\partial \delta_b}|_{\delta_a=\delta_b=0} = \frac{1}{\ln(2)}\left(-\mu_p\ln\left(\frac{\frac{1}{\epsilon}-1}{\theta_t}+1\right) + \frac{(1-\mu_p+\mu_p\theta_t)(\frac{1}{\epsilon}-1)}{(\theta_t)^2(\frac{\frac{1}{\epsilon}-1}{\theta_t}+1)}\right)$$

Similarly to the first case this function is affine with respect to μ_p. Hence, we show it to be positive for the end points $\mu_p = 0$ and $\mu_p = \epsilon$. This in turn shows the function to be positive for μ_p where $0 \leq \mu_p \leq \epsilon$.

Consider $\mu_p = 0$.

$$\frac{\partial e_p(\epsilon,\mu_p,\theta_t,\delta_a,\delta_b)}{\partial \delta_b}|_{\substack{\delta_a=\delta_b=0\\ \mu_p=0}} = \frac{1}{\ln(2)}\left(\frac{(\frac{1}{\epsilon}-1)}{(\theta_t)^2(\frac{\frac{1}{\epsilon}-1}{\theta_t}+1)}\right)$$

This is positive for all $\epsilon > 0$.

Now consider $\mu_p = \epsilon$. Let $x = \frac{\frac{1}{\epsilon}-1}{\theta_t}$ and $f_3(x) = x - \ln(x+1)$.

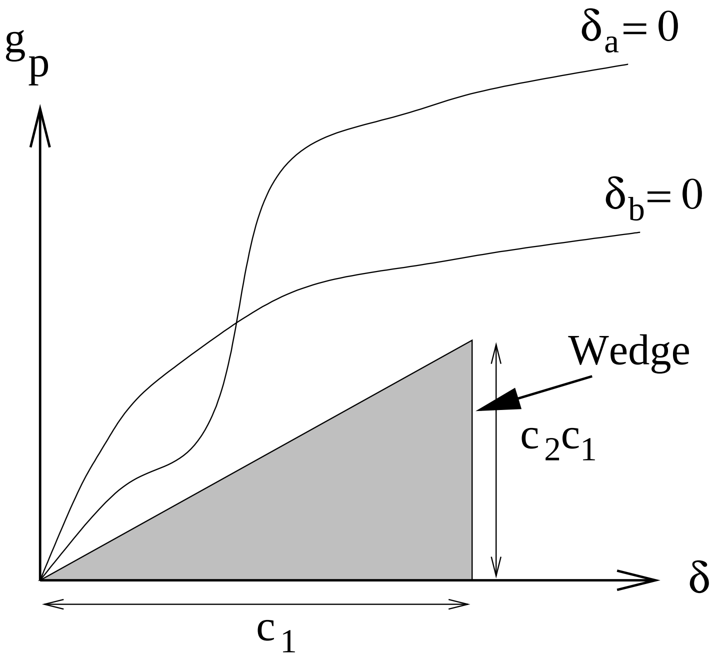


Figure B.3: e_p is shown to be at or above a "wedge" when $\delta_a = 0$ or $\delta_b = 0$.

$$\begin{aligned}
\frac{\partial e_p(\epsilon, \mu_p, \theta_t, \delta_a, \delta_b)}{\partial \delta_b}\Big|_{\substack{\delta_a=\delta_b=0\\ \mu_p=\epsilon}} &= \frac{1}{\ln(2)}\left(-\epsilon \ln\left(\frac{\frac{1}{\epsilon}-1}{\theta_t}+1\right) + \frac{(1-\epsilon+\epsilon\theta_t)(\frac{1}{\epsilon}-1)}{(\theta_t)^2(\frac{\frac{1}{\epsilon}-1}{\theta_t}+1)}\right) \\
&= \frac{\epsilon}{\ln(2)}(x - \ln(x+1)) \\
&= \frac{\epsilon}{\ln(2)} f_3(x)
\end{aligned}$$

Finally, $f_3(x)|_{x=0} = 0$ and $\frac{df_3}{dx} = 1 - \frac{1}{x+1}$ which is positive for all $x > 0$. Hence $\frac{\partial e_p(\epsilon,\mu_p,\theta_t,\delta_a,\delta_b)}{\partial \delta_b}\Big|_{\substack{\delta_a=\delta_b=0\\ \mu_p=\epsilon}} > 0$ for all values of interest.

Step 3 We are required to show that there exists a "wedge" that it is equal to or less than e_p when $\delta_a = 0$ and $\delta_b = \delta$, or when $\delta_b = 0$ and $\delta_a = \delta$. This is diagrammatically shown in Figure B.3. More formally we are required to show there exists constants $c_1 > 0$ and $c_2 > 0$ such that for all $\delta \le c_1$, $e_p(\epsilon, \mu_p, \theta_t, \delta, 0) \ge c_2\delta$ and $e_p(\epsilon, \mu_p, \theta_t, 0, \delta) \ge c_2\delta$. Note, ϵ, μ_p, and θ_t may be considered constants.

To simplify the notation let $f_a(\delta) = e_p(\epsilon, \mu_p, \theta_t, \delta_a, 0)$ and $f_b(\delta) = e_p(\epsilon, \mu_p, \theta_t, 0, \delta_b)$. Let $\gamma_a = \frac{df_a}{d\delta}|_{\delta=0}$ and $\gamma_b = \frac{df_b}{d\delta_b}|_{\delta=0}$. From step 2 we know that $\gamma_a > 0$ and $\gamma_b > 0$. Now let $\gamma = \text{minimum}\{\gamma_a, \gamma_b\}$ and let $c_2 = \frac{1}{2}\gamma$.

Hence, by the definition of the derivative we have $\lim_{x\to 0} \frac{f_a(x) - f_a(0)}{x - 0} = \gamma_a$ which simplifies to $\lim_{x\to 0} \frac{f_a(x)}{x} = \gamma_a$. Correspondingly, $\lim_{x\to 0} \frac{f_b(x)}{x} = \gamma_b$. Now from the definition of a limit[1] we

[1]Normally the variables ϵ and δ would be use in these limit proofs, however, as they have been used extensively in this chapter for the error parameter and error from the correct hypothesis, they are replaced with ζ and η

have that for every $\zeta > 0$ there exists a $\eta_a > 0$ where $|\frac{f_a(x)}{x} - \gamma_a| < \zeta$ for all values of x such that $|x| < \eta_a$ and $x \neq 0$. Let $\zeta = c_2$ we know there there exist a $\eta_a > 0$ where $|\frac{f_a(x)}{x} - \gamma_a| < \frac{1}{2}\gamma_a$ for all values of x such that $|x| < \eta_a$.

$$\begin{aligned}
&|\frac{f_a(x)}{x} - \gamma_a| < \frac{1}{2}\gamma_a \\
\Rightarrow &-\frac{f_a(x)}{x} + \gamma_a < \frac{1}{2}\gamma_a \\
\Leftrightarrow &-\frac{f_a(x)}{x} < -\frac{1}{2}\gamma_a \\
\Leftrightarrow &f_a(x) > \frac{1}{2}\gamma_a x \\
\Rightarrow &f_a(x) \geq \frac{1}{2}\gamma_a x
\end{aligned}$$

Similarly for f_b there exists a $\eta_b > 0$ such that $f_b(x) \geq \frac{1}{2}\gamma_b x$ for all $x < \eta_b$. Let $c_1 = \text{minimum}\{\eta_a, \eta_b, c_3\}$. Hence, $f_a(x) \geq \frac{1}{2}\gamma_a x \geq c_2 x$ and $f_b(x) \geq \frac{1}{2}\gamma_b x \geq c_2 x$ for all $x < c_1$. Note, c_3 is included in the minimum, since it is only when $\delta \leq c_3$ that we know that one of the end points will be the minimum of e_p when δ is fixed.

■

Lemma B.0.3 *For any $c_1, c_2 \in \mathbb{R}$ where $c_1 > 0$, and $c_2 > 0$ the following sum is finite.*

$$\sum_{i=1}^{\infty} e^{-c_1 i^{c_2}}$$

PROOF:

There exists m such that for all $i > m$:

$$\begin{aligned}
&c_1 i^{c_2} > 2\ln(i) \\
\Rightarrow &-c_1 i^{c_2} < -2\ln(i) \\
\Rightarrow &e^{-c_1 i^{c_2}} < \frac{1}{i^2}
\end{aligned}$$

As the series $\frac{1}{i^2}$ converges the series in question will also converge.

■

Lemma B.0.4 *Suppose $0 \leq \epsilon < 1$, $0 \leq \mu_p \leq \epsilon$, $0 < \theta_t \leq 1$, $0 \leq \delta_a \leq \theta_t$, $0 \leq \delta_b \leq 1 - \theta_t$, and $0 < \omega \leq$ minimum$\{\theta_t - \delta_a, 1 - \theta_t - \delta_b\}$. Then $e_p(\epsilon, \mu_p, \theta_t, \delta_a, \delta_b) < e_p(\epsilon, \mu_p, \theta_t, \delta_a + \omega, \delta_b + \omega)$ where*

$$\begin{aligned}
e_p(\epsilon, \mu_p, \theta_t, \delta_a, \delta_b) = &(1 - \mu_p + \mu_p\theta_t)\lg(\frac{1-\epsilon}{\theta_t} + \epsilon) \\
&- ((1-\mu_p)(1 - \frac{\delta_a}{\theta_t}) + \mu_p(\theta_t - \delta_a) + \mu_p\delta_b))\lg(\frac{1-\epsilon}{\theta_t - \delta_a + \delta_b} + \epsilon) \\
&- ((1-\mu_p)\frac{\delta_a}{\theta_t} + \mu_p(\delta_a - \delta_b))\lg(\epsilon)
\end{aligned}$$

PROOF: Let $D = e_p(\epsilon, \mu_p, \theta_t, \delta_a + \omega, \delta_b + \omega) - e_p(\epsilon, \mu_p, \theta_t, \delta_a, \delta_b)$. We now show that $D > 0$ for $\omega > 0$.

respectively.

First, consider the case when $\epsilon = 0$. Hence $\mu_p = 0$ and:

$$
\begin{aligned}
D &= \frac{\omega}{\theta_t}\left(\lg(\frac{1}{\theta_t - \delta_a + \delta_b}) - \lg(0)\right) \\
&= \infty \\
&> 0
\end{aligned}
$$

Second, consider the case when $\epsilon > 0$. Then:

$$
\begin{aligned}
D &= \frac{\omega(1-\mu_p)}{\theta_t}\left(\lg(\frac{\frac{1}{\epsilon}-1}{\theta_t - \delta_a + \delta_b} + 1)\right) \\
&> 0 \quad \text{as } \omega > 0,\ (1-\mu_p) > 0, \text{ and } \lg(\frac{\frac{1}{\epsilon}-1}{\theta_t - \delta_a + \delta_b} + 1) > 0
\end{aligned}
$$

■

Lemma B.0.5 *Suppose* $0 \le \epsilon < 1$, $0 \le \mu_p \le \epsilon$, $0 < \theta_t \le 1$, $\delta_a = 0$, $0 \le \delta_b \le 1 - \theta_t$, *and* $0 < \omega \le 1 - \theta_t - \delta_b$. *Then* $e_p(\epsilon, \mu_p, \theta_t, 0, \delta_b) < e_p(\epsilon, \mu_p, \theta_t, 0, \delta_b + \omega)$ *where*

$$
\begin{aligned}
e_p(\epsilon, \mu_p, \theta_t, \delta_a, \delta_b) =&(1 - \mu_p + \mu_p\theta_t)\lg(\frac{1-\epsilon}{\theta_t} + \epsilon) \\
&- ((1-\mu_p)(1 - \frac{\delta_a}{\theta_t}) + \mu_p(\theta_t - \delta_a) + \mu_p\delta_b))\lg(\frac{1-\epsilon}{\theta_t - \delta_a + \delta_b} + \epsilon) \\
&- ((1-\mu_p)\frac{\delta_a}{\theta_t} + \mu_p(\delta_a - \delta_b))\lg(\epsilon)
\end{aligned}
$$

Proof: Let $D = e_p(\epsilon, \mu_p, \theta_t, 0, \delta_b + \omega) - e_p(\epsilon, \mu_p, \theta_t, 0, \delta_b)$. We now show that $D > 0$ for $\omega > 0$.

First, consider the case when $\epsilon = 0$. Hence $\mu_p = 0$ and:

$$
\begin{aligned}
D &= \lg(\frac{1}{\theta_t + \delta_b}) - \lg(\frac{1}{\theta_t + \delta_b + \omega}) \\
&> 0 \quad \text{as } \theta_t + \delta_b < \theta_t + \delta_b + \omega
\end{aligned}
$$

Second, consider the case when $\epsilon > 0$.

$$
\begin{aligned}
D =&(1 - \mu_p + \mu_p\theta_t + \mu_p\delta_b)\lg(\frac{\frac{1}{\epsilon}-1}{\theta_t + \delta_b} + 1) - \\
&(1 - \mu_p + \mu_p\theta_t + \mu_p(\omega + \delta_b))\lg(\frac{\frac{1}{\epsilon}-1}{\theta_t + \delta_b + \omega} + 1)
\end{aligned}
$$

Note that $D|_{\omega=0} = 0$ and by showing that the gradient of D with respect to ω is positive for $\omega > 0$, we show that $D > 0$ for $\omega > 0$, as required.

$$
\frac{\partial D}{\partial \omega} = \frac{1}{\ln(2)}\left(\frac{(1 - \mu_p + \mu_p\theta_t + \mu_p(\delta_b + \omega))(\frac{1}{\epsilon} - 1)}{(\theta_t + \delta_b + \omega)^2(\frac{\frac{1}{\epsilon}-1}{\theta_t+\delta_b+\omega} + 1)} - \mu_p\ln(\frac{\frac{1}{\epsilon}-1}{\theta_t + \delta_b + \omega} + 1)\right)
$$

The partial derivative $\frac{\partial D}{\partial \omega}$ is affine with respect to μ_p, so by showing it to be positive at $\mu_p = 0$

and $\mu_p = \epsilon$, it will be proved to be positive for all $0 \le \mu_p \le \epsilon$. Consider $\mu_p = 0$.

$$\frac{\partial D}{\partial \omega}|_{\mu_p=0} = \frac{(\frac{1}{\epsilon}-1)}{\ln(2)(\theta_t+\delta_b+\omega)^2(\frac{\frac{1}{\epsilon}-1}{\theta_t+\delta_b+\omega}+1)}$$
$$> 0$$

Consider $\mu_p = \epsilon$. Let $f(x,\mu_p)$ be equal to $\frac{\partial D}{\partial \omega}$ where $\epsilon = \mu_p$ and $x = \theta_t + \delta_b + \omega$. Note, $0 < x \le 1$ and $0 < \mu_p < 1$.

$$f(x,\mu_p) = \frac{\mu_p x \ln(\frac{\mu_p x}{1-\mu_p+\mu_p x}) + 1 - \mu_p}{x\ln(2)}$$

$$\frac{\partial f}{\partial \mu_p} = \frac{x\ln(\frac{\mu_p x}{1-\mu_p+\mu_p x}) - 1 + x + \frac{\mu_p x(1-x)}{1-\mu_p+\mu_p x}}{x\ln(2)}$$

$$\frac{\partial^2 f}{\partial \mu_p^2} = \frac{\frac{2(1-x)x}{1-\mu_p+\mu_p x} + \frac{\mu_p(x-1)^2 x}{(1-\mu_p+\mu_p x)^2} + \frac{x}{m}}{x\ln(2)}$$
$$> 0$$

As $f(x,\mu_p)|_{\mu_p=0} = 0$, $\frac{\partial f}{\partial \mu_p}|_{\mu_p=0} = 0$ and f is concave up for $0 < \mu_p < 1$, the function f is positive for any $0 < \mu_p < 1$ and any $0 < x \le 1$.

So the partial derivative $\frac{\partial D}{\partial \omega}$ is positive and hence $D > 0$ for $\omega > 0$, as required. ■

Lemma B.0.6 *Suppose* $0 \le \epsilon < 1$, $0 \le \mu_p \le \epsilon$, $0 < \theta_t \le 1$, $0 \le \delta_a \le \theta_t$, $\delta_b = 0$, *and* $0 < \omega \le \theta_t - \delta_a$. *Then* $e_p(\epsilon,\mu_p,\theta_t,\delta_a,0) < e_p(\epsilon,\mu_p,\theta_t,\delta_a+\omega,0)$ *where*

$$\begin{aligned} e_p(\epsilon,\mu_p,\theta_t,\delta_a,\delta_b) =& (1-\mu_p+\mu_p\theta_t)\lg(\frac{1-\epsilon}{\theta_t}+\epsilon) \\ &- ((1-\mu_p)(1-\frac{\delta_a}{\theta_t}) + \mu_p(\theta_t-\delta_a) + \mu_p\delta_b))\lg(\frac{1-\epsilon}{\theta_t-\delta_a+\delta_b}+\epsilon) \\ &- ((1-\mu_p)\frac{\delta_a}{\theta_t} + \mu_p(\delta_a-\delta_b))\lg(\epsilon) \end{aligned}$$

Proof: Let $D = e_p(\epsilon,\mu_p,\theta_t,\delta_a+\omega,0) - e_p(\epsilon,\mu_p,\theta_t,\delta_a,0)$. We now show that $D > 0$ for $\omega > 0$.

First, consider the case when $\epsilon = 0$. Hence $\mu_p = 0$ and:

$$\begin{aligned} D &= (1-\frac{\delta_a}{\theta_t})\lg(\frac{1}{\theta_t-\delta_a}) - (1-\frac{\delta_a+\omega}{\theta_t})\lg(\frac{1}{\theta_t-\delta_a-\omega}) - \frac{\omega}{\theta_t}\lg(0) \\ &= \infty \\ &> 0 \end{aligned}$$

Second, consider the case when $\epsilon > 0$.

$$D = (1 - \mu_p + \mu_p\theta_t - \delta_a(\frac{1-\mu_p}{\theta_t} + \mu_p)) \lg(\frac{\frac{1}{\epsilon} - 1}{\theta_t - \delta_a} + 1) -$$
$$(1 - \mu_p + \mu_p\theta_t - (\omega + \delta_a)(\frac{1-\mu_p}{\theta_t} + \mu_p)) \lg(\frac{\frac{1}{\epsilon} - 1}{\theta_t - \delta_a - \omega} + 1)$$

Note that $D|_{\omega=0} = 0$. By showing that the gradient of D with respect to ω is positive for $\omega > 0$, we prove that $D > 0$ for $\omega > 0$, as required.

Let $x = \theta_t - \delta_a - \omega$ (hence $0 < x < 1$). Then:

$$\begin{aligned}\frac{\partial D}{\partial \omega} &= \frac{1}{\ln(2)}\Bigg((\frac{1-\mu_p}{\theta_t} + \mu_p)\ln(\frac{\frac{1}{\epsilon} - 1}{\theta_t - \delta_a - \omega} + 1) - \\ &\qquad \frac{(1 - \mu_p + \mu_p\theta_t - (\delta_a + \omega)(\frac{1-\mu_p}{\theta_t} + \mu_p))(\frac{1}{\epsilon} - 1)}{(\theta_t - \delta_a - \omega)^2(\frac{\frac{1}{\epsilon}-1}{\theta_t - \delta_a - \omega} + 1)}\Bigg) \\ &= \frac{(1 - \mu_p + \mu_p\theta_t)(-(1 - \epsilon + \epsilon x)\ln(\frac{\epsilon x}{1-\epsilon+\epsilon x}) - 1 + \epsilon)}{\theta_t(1 - \epsilon + \epsilon x)\ln(2)}\end{aligned}$$

Let $f(\epsilon, x) = -(1 - \epsilon + \epsilon x)\ln(\frac{\epsilon x}{1-\epsilon+\epsilon x}) - 1 + \epsilon$. Now $f(\epsilon, x)|_{\epsilon=1} = 0$ and we can calculate the first and second derivatives of f with respect to ϵ (note the variable x is not dependent on ϵ).

$$\frac{\partial f}{\partial \epsilon} = (1 - x)\ln(\frac{x\epsilon}{1 - \epsilon + \epsilon x}) + x - \frac{1 - \epsilon + \epsilon x}{\epsilon}$$

$$\begin{aligned}\frac{\partial^2 f}{\partial \epsilon^2} &= \frac{(x-1)^2}{1 - \epsilon + \epsilon x} + \frac{1 - \epsilon + \epsilon x}{\epsilon^2} + \frac{2(1-x)}{\epsilon} \\ &> 0\end{aligned}$$

As $\frac{\partial f}{\partial \epsilon}|_{\epsilon=1} = 0$ and the function is concave up for $0 \leq \epsilon < 1$, f is clearly positive for the cases of interest.

Moreover, $\frac{\partial D}{\partial \omega}$ is positive as each of the terms in both the denominator and enumerator are positive. It follows that $D > 0$ for $\omega > 0$, as required. ■

Lemma B.0.7 *Suppose $c > 0$, $0 \leq \epsilon < 1$, $0 \leq \mu_p \leq \epsilon$, $0 < \theta_t \leq 1$, $0 \leq \delta \leq \max(\theta_t, 1 - \theta_t)$, and $0 \leq \omega \leq \max(\theta_t, 1 - \theta_t) - \delta$. Then:*

$$(\forall \delta_a \in [\max(0, \delta - (1 - \theta_t)), \min(\theta_t, \delta)])\ e_p(\epsilon, \mu_p, \theta_t, \delta_a, \delta - \delta_a) \geq c$$
$$\Rightarrow$$
$$(\forall \delta_a \in [\max(0, \delta + \omega - (1 - \theta_t)), \min(\theta_t, \delta + \omega)])\ e_p(\epsilon, \mu_p, \theta_t, \delta_a, \delta + \omega - \delta_a) \geq c$$

where

$$
\begin{aligned}
e_p(\epsilon,\mu_p,\theta_t,\delta_a,\delta_b) =&(1-\mu_p+\mu_p\theta_t)\lg(\frac{1-\epsilon}{\theta_t}+\epsilon)\\
&-((1-\mu_p)(1-\frac{\delta_a}{\theta_t})+\mu_p(\theta_t-\delta_a)+\mu_p\delta_b))\lg(\frac{1-\epsilon}{\theta_t-\delta_a+\delta_b}+\epsilon)\\
&-((1-\mu_p)\frac{\delta_a}{\theta_t}+\mu_p(\delta_a-\delta_b))\lg(\epsilon)
\end{aligned}
$$

Proof: The proof is divided into three cases which are then combined to yield the required result.

Case 1

$$
\begin{aligned}
&((\forall\delta_a\in[\max(0,\delta-(1-\theta_t)),\min(\theta_t,\delta)])\\
&\qquad e_p(\epsilon,\mu_p,\theta_t,\delta_a,\delta-\delta_a)\geq c\\
\Rightarrow&(\forall\delta_a\in[\max(0,\delta-(1-\theta_t)+\frac{1}{2}\omega),\min(\theta_t-\frac{1}{2}\omega,\delta)])\\
&\qquad e_p(\epsilon,\mu_p,\theta_t,\delta_a+\frac{1}{2}\omega,\delta-\delta_a+\frac{1}{2}\omega)\geq c\quad\text{(Lemma B.0.4)}\\
\Rightarrow&(\forall\delta'_a\in[\max(\frac{1}{2}\omega,\delta-(1-\theta_t)+\omega),\min(\theta_t,\delta+\frac{1}{2}\omega)])\\
&\qquad e_p(\epsilon,\mu_p,\theta_t,\delta'_a,\delta+\omega-\delta'_a)\geq c\quad\text{where }\delta'_a=\delta_a+\frac{1}{2}\omega
\end{aligned}
$$

Case 2

$$
\begin{aligned}
&((\forall\delta_a\in[\max(0,\delta-(1-\theta_t)),\min(\theta_t,\delta)])\\
&\qquad e_p(\epsilon,\mu_p,\theta_t,\delta_a,\delta-\delta_a)\geq c)\wedge\delta\leq 1-\theta_t-\frac{1}{2}\omega\\
\Rightarrow&(\forall\delta_a\in[0,\min(\theta_t,\delta)])\\
&\qquad e_p(\epsilon,\mu_p,\theta_t,\delta_a,\delta-\delta_a)\geq c\\
\Rightarrow&e_p(\epsilon,\mu_p,\theta_t,0,\delta)\geq c\\
\Rightarrow&(\forall x\in[\max(0,\omega-2\theta_t),\min(\omega,2(1-\theta_t-\delta)-\omega)])\\
&\qquad e_p(\epsilon,\mu_p,\theta_t,0,\delta+x)\geq c\quad\text{(Lemma B.0.5)}\\
\Rightarrow&(\forall x\in[\max(0,\omega-2\theta_t),\min(\omega,2(1-\theta_t-\delta)-\omega)],y=\frac{1}{2}(\omega-x))\\
&\qquad e_p(\epsilon,\mu_p,\theta_t,y,\delta+x+y)\geq c\quad\text{(Lemma B.0.4)}\\
\Rightarrow&(\forall\delta_a\in[\max(0,\omega-(1-\theta_t-\delta)),\min(\frac{1}{2}\omega,\theta_t)])\\
&\qquad e_p(\epsilon,\mu_p,\theta_t,\delta_a,\delta+\omega-\delta_a)\geq c\quad\text{where }\delta_a=\frac{1}{2}(\omega-x)
\end{aligned}
$$

Now, if $\delta>1-\theta_t-\frac{1}{2}\omega$ then $[\max(0,\omega-(1-\theta_t-\delta)),\min(\frac{1}{2}\omega,\theta_t)]$ will be empty. Hence,

$$
\begin{gathered}
(\forall\delta_a\in[\max(0,\delta-(1-\theta_t)),\min(\theta_t,\delta)])\ e_p(\epsilon,\mu_p,\theta_t,\delta_a,\delta-\delta_a)\geq c\\
\Rightarrow\\
(\forall\delta_a\in[\max(0,\omega-(1-\theta_t-\delta)),\min(\tfrac{1}{2}\omega,\theta_t)])\ e_p(\epsilon,\mu_p,\theta_t,\delta_a,\delta+\omega-\delta_a)\geq c
\end{gathered}
$$

Case 3

$$
\begin{aligned}
&(\forall \delta_a \in [\max(0, \delta - (1-\theta_t)), \min(\theta_t, \delta)]) \\
&\qquad e_p(\epsilon, \mu_p, \theta_t, \delta_a, \delta - \delta_a) \geq c) \wedge \delta \leq \theta_t - \frac{1}{2}\omega \\
\Rightarrow &(\forall \delta_a \in [\max(0, \delta - (1-\theta_t)), \delta]) \\
&\qquad e_p(\epsilon, \mu_p, \theta_t, \delta_a, \delta - \delta_a) \geq c \\
\Rightarrow &e_p(\epsilon, \mu_p, \theta_t, \delta, 0) \geq c \\
\Rightarrow &(\forall x \in [\max(0, \omega - 2(1-\theta_t)), \min(\omega, 2(\theta_t - \delta) - \omega)]) \\
&\qquad e_p(\epsilon, \mu_p, \theta_t, \delta + x, 0) \geq c \quad \text{(Lemma B.0.6)} \\
\Rightarrow &(\forall x \in [\max(0, \omega - 2(1-\theta_t)), \min(\omega, 2(\theta_t - \delta) - \omega)], y = \frac{1}{2}(\omega - x)) \\
&\qquad e_p(\epsilon, \mu_p, \theta_t, \delta + x + y, y) \geq c \quad \text{(Lemma B.0.4)} \\
\Rightarrow &(\forall \delta_a \in [\max(\delta + \frac{1}{2}\omega, \omega - (1 - \theta_t - \delta)), \min(\delta + \omega, \theta_t)]) \\
&\qquad e_p(\epsilon, \mu_p, \theta_t, \delta_a, \delta + \omega - \delta_a) \geq c \quad \text{where } \delta_a = \delta + \frac{1}{2}\omega + \frac{1}{2}x
\end{aligned}
$$

If $\delta > \theta_t - \frac{1}{2}\omega$ then $[\max(\delta + \frac{1}{2}\omega, \omega - (1 - \theta_t - \delta)), \min(\delta + \omega, \theta_t)]$ is empty and hence:

$$
\begin{gathered}
(\forall \delta_a \in [\max(0, \delta - (1-\theta_t)), \min(\theta_t, \delta)])\ e_p(\epsilon, \mu_p, \theta_t, \delta_a, \delta - \delta_a) \geq c \\
\Rightarrow \\
(\forall \delta_a \in [\max(\delta + \tfrac{1}{2}\omega, \omega - (1 - \theta_t - \delta)), \min(\delta + \omega, \theta_t)])\ e_p(\epsilon, \mu_p, \theta_t, \delta_a, \delta + \omega - \delta_a) \geq c
\end{gathered}
$$

Now we may combine the three cases and show the required result.

$$
\begin{aligned}
&(\forall \delta_a \in [\max(0, \delta - (1-\theta_t)), \min(\theta_t, \delta)])\ e_p(\epsilon, \mu_p, \theta_t, \delta_a, \delta - \delta_a) \geq c \\
\Rightarrow &[(\forall \delta_a \in [\max(0, \delta + \omega - (1-\theta_t)), \min(\theta_t, \frac{1}{2}\omega)])\ e_p(\epsilon, \mu_p, \theta_t, \delta_a, \delta + \omega - \delta_a) \geq c] \wedge \\
&[(\forall \delta_a \in [\max(\frac{1}{2}\omega, \delta + \omega - (1-\theta_t)), \min(\theta_t, \delta + \frac{1}{2}\omega)])\ e_p(\epsilon, \mu_p, \theta_t, \delta_a, \delta + \omega - \delta_a) \geq c] \wedge \\
&[(\forall \delta_a \in [\max(\delta + \frac{1}{2}\omega, \delta + \omega - (1-\theta_t)), \min(\theta_t, \delta + \omega)])\ e_p(\epsilon, \mu_p, \theta_t, \delta_a, \delta + \omega - \delta_a) \geq c] \\
\Rightarrow &(\forall \delta_a \in [\max(0, \delta + \omega - (1-\theta_t)), \min(\theta_t, \delta + \omega)])\ e_p(\epsilon, \mu_p, \theta_t, \delta_a, \delta + \omega - \delta_a) \geq c
\end{aligned}
$$

■

Lemma B.0.8 *Given $\alpha \in (0,1)$, $m \geq 1$, and $i \geq 1$:*

$$\frac{m}{i^{2\alpha}} + \frac{i^2}{m} \geq m^{\frac{1-\alpha}{1+\alpha}}$$

Proof: The proof is split into two cases.

Case 1 $i \leq m^{\frac{1}{1+\alpha}}$

$$
\begin{aligned}
\frac{m}{i^{2\alpha}} + \frac{i^2}{m} &\geq \frac{m}{i^{2\alpha}} \\
&\geq \frac{m}{m^{\frac{2\alpha}{1+\alpha}}} \\
&= m^{1-\frac{2\alpha}{1+\alpha}} \\
&= m^{\frac{1-\alpha}{1+\alpha}}
\end{aligned}
$$

Case 2 $i > m^{\frac{1}{1+\alpha}}$

$$
\begin{aligned}
\frac{m}{i^{2\alpha}} + \frac{i^2}{m} &\geq \frac{i^2}{m} \\
&\geq \frac{m^{\frac{2}{1+\alpha}}}{m} \\
&= m^{\frac{2}{1+\alpha}-1} \\
&= m^{\frac{1-\alpha}{1+\alpha}}
\end{aligned}
$$

■

Lemma B.0.9 *Given $m \geq 1$, and $i \geq 2$:*

$$
\frac{m}{(\lg(i))^2} + \frac{i}{(\lg(i))^2} \geq m^{\frac{1}{4}} + i^{\frac{1}{4}}
$$

Proof:

$$
\begin{aligned}
\frac{m}{(\lg(i))^2} + \frac{i}{(\lg(i))^2} &\geq \frac{m}{(\lg(i))^2} + i^{\frac{1}{2}} \qquad (\text{As } \frac{i^{\frac{1}{2}}}{(\lg(i))^2} \geq 1) \\
&\geq \frac{m}{(\lg(i))^2} + i^{\frac{1}{4}} + i^{\frac{1}{4}} \\
&\geq m^{\frac{1}{4}} + i^{\frac{1}{4}} \qquad (\text{As } m^{\frac{1}{4}} \leq \frac{m}{(\lg(i))^2} + i^{\frac{1}{4}})
\end{aligned}
$$

Note the final step in the above inequality may be simply shown by dividing the proof into two cases, $i \geq m$ and $m < i$ in a similar process to the proof for Lemma B.0.8.

■

Appendix C

Meta-Knowledge

Many ILP systems, such as Golem, Foil, and Flip, take advantage of user supplied meta-knowledge to restrict the hypothesis space. This meta-knowledge can be in the form of type information about arguments in the predicate being learned, or it can be information about whether a certain argument in the predicate is functionally dependent on the other arguments (supplied as mode information). This meta-knowledge is explicitly supplied to an ILP system in addition to the data.

The appendix argues that in many cases the meta-knowledge can be extracted directly from the raw data. Three algorithms are presented that learn type, mode, and symmetric meta-knowledge from data. The type and mode learning algorithms are directly incorporated into Lime, improving its performance and usability. However, as this meta-knowledge may be learnt separately and used in a variety of ILP systems it is appropriate to investigate this approach separately from Lime. The algorithms can be incorporated into existing ILP systems in the form of a preprocessor that obviates the need for a user to explicitly provide this information. In many cases, the algorithms can extract meta-knowledge that the user is unaware of, but which can be used by the ILP system to restrict the hypothesis space.

Empirical evidence for the effectiveness of these algorithms is demonstrated by incorporating them into Foil. Finally, the biasing effects of the different kinds of meta-knowledge on the hypothesis space are discussed.

C.1 Introduction

The size of the hypothesis space is usually huge. In order to discover a correct hypothesis feasibly, this space must be structured and searched efficiently. Many ILP systems provide extra information or meta-knowledge [55, 59] about the relation being learned. This information renders a large number of hypotheses in the space incompatible, and the system can safely exclude them from the search space. This compacts or biases the search space and improves the efficiency of the ILP system.

Current ILP systems employ a variety of meta-knowledge to bias the hypothesis space. This extra information includes, rule models [51], functional/mode constraints [11, 90, 55, 122, 44], predicate sets [10], and type constraints [122, 44].

	Mode	Type	Symmetry	Exclusive Opposite
MISST [55]	√	√	√	√
MARKUS [44]	√	√	√	
PROGOL [94]	√	√		
MIS [122]	√	√		
GOLEM [90]	√			
FOIL [25]	√	√		
FLIP [11]	√			
MOBAL [79]		√		

Table C.1: ILP Systems using meta knowledge

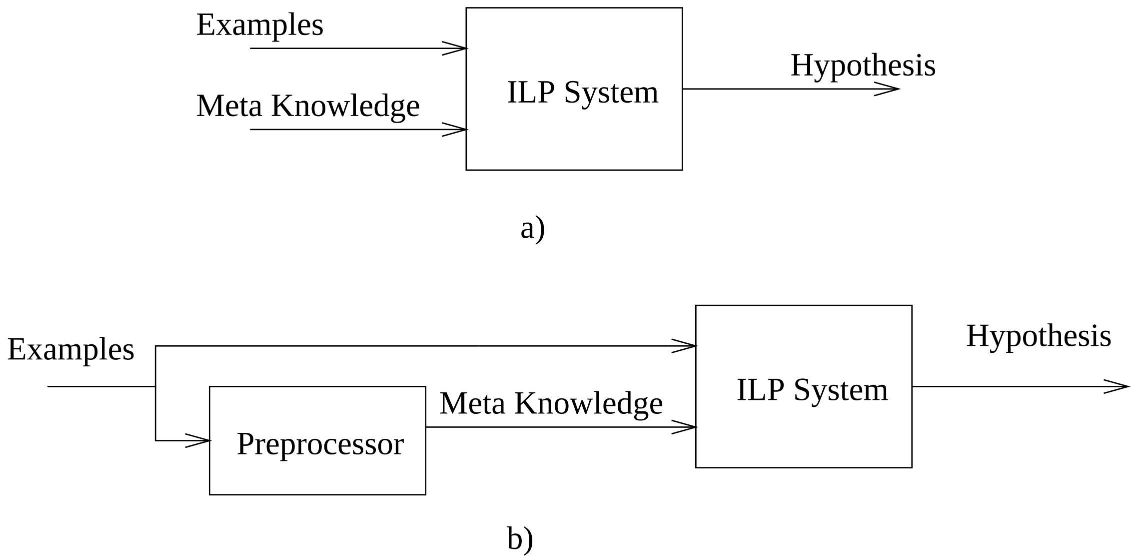


Figure C.1: (a) ILP System, (b) ILP System with property generator

Table C.1 shows the biasing properties employed by different ILP systems. All these systems require that the user explicitly provides this additional information. In some cases, the information is necessary to the operation of the system; in others, the biasing information is optional, and when added improves the performance.

Several of these properties are implicitly contained in the raw data. An ILP system may be modified by attaching a preprocessor of the data that acts as a property generator, as shown in Figure C.1. Preprocessing of the raw data can yield different kinds of meta-knowledge, making it unnecessary for the user to explicitly provide it. This yields systems that learn with less assistance from the user. Moreover, if the user is not aware of certain properties of the data, or is unable to provide them, then this technique allows the system to perform more efficiently, and in some cases succeed in learning where it was previously unable to learn. Additionally, the same preprocessor can be used to learn properties of the background knowledge. This further improves the efficiency of the ILP system by creating a strong bias on the hypothesis space.

As noted earlier, additional information to an ILP system may include rule models, predicate sets, type constraints, functional/mode constraints, one-to-one and onto relations, symmetric properties, anti-symmetric properties, exclusive binary predicates, transitive binary predicates, and reflexive binary predicates. This list is in no way exhaustive. Information such as rule models and predicate sets can not be learned by examining the examples of the relation. However, other properties, such as type, mode, symmetric, anti-symmetric, exclusive, transitive, and reflexive properties are implicitly present within the data. This appendix describes a preprocessor that extracts type, functional dependency, and symmetry information from data. We briefly illustrate these properties next.

The relation for the length of a list has two arguments, the first is of type *List* and the second is of type *Integer*. This information may be described by a type declaration $Type(length(List, Integer))$. The relation for addition can be expressed as $add(t_1, t_2, t_3)$, where t_3 is the sum of t_1 and t_2. Now, t_3 is functionally dependent on t_1, t_2. That is, when t_1 and t_2 are bound, t_3 will also be bound. This information could be given to the ILP system in the form of a mode declaration $Mode(add(+, +, -))$. Also, *addition* is symmetric in its first two arguments; this information may be given to an ILP system in the form of a symmetry declaration $Symmetry(add(*, *, -))$.

The effectiveness of the preprocessor is demonstrated by combining it with FOIL [25]. We now proceed as follows. In Section C.2, we illustrate the working of the algorithms for extracting type, mode, and symmetric information with the help of an example. In Section C.3, we present and discuss experimental results demonstrating the speedup achieved as a result of the use of the preprocessor with FOIL. Section C.4 summarizes the complexity of the algorithms and discusses its theoretical properties. Finally, in Section C.5, we briefly touch upon the usefulness of this approach in the design of an incremental ILP system that dynamically modifies its hypothesis space based on feedback from a preprocessor that inspects the data for meta-knowledge.

C.2 An Illustration of the Approach

We illustrate the proposed preprocessing technique in the context of a system that is required to learn an intensional definition of the predicate *merge*. This predicate, which takes three arguments, asserts that the third argument is a sorted list formed by merging the first two arguments that are also sorted lists. So, for example $merge([1, 4, 6], [3, 6], [1, 3, 4, 6, 6])$ is a positive example of the *merge* predicate and $merge([1, 2], [], [2, 1])$ is a negative example. Suppose the system is also provided with background knowledge about the predicates *less*, *less-eq*, and *comp*. Predicates *less* and *less-eq* have the usual meaning over numbers; the predicate *comp* is true if the second and the third arguments are the *head* and *tail* of the list in the first argument. We assume that the background knowledge about these predicates is also provided extensionally as shown in Table C.2. We now demonstrate how meta knowledge about type, mode, and symmetry can be extracted purely by examining the data. The three algorithms are sketched in Figure C.2.

Below, the algorithms for functional dependency and symmetry extract correct information in the incremental (limiting) sense from complete data or under the closed world assumption. In batch mode, without the closed world assumption, the correctness of the information extracted depends on the "density" of the data. The algorithm for extracting type information, however, always produces the the most specific type assignment implied by the given data.

E^+	*merge*([], [], [1]).	*less*(1, 2).
merge([], [], []).	*merge*([1], [1], []).	*less-eq*(1, 2).
merge([], [1], [1]).	*merge*([], [2], [1]).	*less-eq*(1, 1).
merge([], [2], [2]).	*merge*([], [1, 2], [1, 1, 2]).	*less-eq*(2, 2).
merge([], [1, 2], [1, 2]).	*merge*([1], [2], [1]).	*comp*([1], 1, []).
merge([1], [], [1]).	*merge*([1], [1, 2], [1, 2]).	*comp*([2], 2, []).
merge([1], [1], [1, 1]).	*merge*([2], [], []).	*comp*([1, 1], 1, [1]).
merge([1], [2], [1, 2]).	*merge*([2], [1], [2, 1]).	*comp*([1, 2], 1, [2]).
merge([1], [1, 2], [1, 1, 2]).	*merge*([2], [2], [2]).	*comp*([2, 1], 2, [1]).
merge([2], [], [2]).	*merge*([1, 2], [], [2, 1]).	*comp*([2, 2], 2, [2]).
merge([2], [1], [1, 2]).	*merge*([1, 2], [1], [1, 2, 1]).	*comp*([1, 1, 1], 1, [1, 1]).
merge([2], [2], [2, 2]).	*merge*([1, 2], [2], [2, 1]).	*comp*([1, 1, 2], 1, [1, 2]).
merge([2], [1, 2], [1, 2, 2]).	*merge*([1, 2], [1, 2], [1, 2]).	*comp*([1, 2, 1], 1, [2, 1]).
merge([1, 2], [], [1, 2]).		⋮
merge([1, 2], [1], [1, 1, 2]).		
merge([1, 2], [2], [1, 2, 2]).		
merge([1, 2], [1, 2], [1, 1, 2, 2]).		*comp*([2, 2, 1], 2, [2, 1]).
E^-	B	*comp*([2, 2, 2], 2, [2, 2]).

Table C.2: Raw information for learning *merge*

C.2.1 Type

A type assignment that is consistent with the predicate being learned and the predicates in the background knowledge can be found in one pass of the positive examples, negative examples, and the background knowledge (i.e., $E^+ \cup E^- \cup B$). This may proceed as follows. Initially, set the types associated with each argument of each predicate to a different type. This may be represented as:

$$Type(merge(\tau_1, \tau_2, \tau_3), less(\tau_4, \tau_5), less\text{-}eq(\tau_6, \tau_7), comp(\tau_8, \tau_9, \tau_{10}))$$

Then begin to scan each member of $E^+ \cup E^- \cup B$. An inspection of the very first fact, *merge*([], [], []), implies that each argument of *merge* must have the same type, as the set of types partitions the Herbrand universe and each argument contains the same element [].[1] The type assignment will now be updated to:

$$Type(merge(\tau_1, \tau_1, \tau_1), less(\tau_4, \tau_5), less\text{-}eq(\tau_6, \tau_7), comp(\tau_8, \tau_9, \tau_{10}))$$

The scan will pass over *less*(1, 2) having no effect on the type assignment as there is no evidence to show that 1 and 2 are of the same type. When *less-eq*(1, 2) is reached, and the same elements are seen as in the previous fact *less*(1, 2), it can be inferred that both the predicates must have the same type vector. Thus, the type assignment will now be updated to:

$$Type(merge(\tau_1, \tau_1, \tau_1), less(\tau_4, \tau_5), less\text{-}eq(\tau_4, \tau_5), comp(\tau_8, \tau_9, \tau_{10}))$$

[1] Note, the *type* extraction algorithm as described does not distinguish between types, one of which is a subset of the other; for example, the set of natural numbers and the set of integers.

This process is continued until $E^+ \cup E^- \cup B$ is completely scanned. The final type assignment will be:

$$Type(merge(\tau_1, \tau_1, \tau_1), less(\tau_4, \tau_4), less\text{-}eq(\tau_4, \tau_4), comp(\tau_1, \tau_4, \tau_1))$$

This type meta-knowledge would restrict the search space by excluding clauses that are inconsistent with this type assignment. For example, a clause such as

$$merge(X_1, X_2, X_3) \leftarrow comp(X_1, X_4, X_3), less(X_4, X_1)$$

is inconsistent with the induced type assignment, and hence can be excluded from the search space, thereby improving the performance of the system. An outline of this approach is given in Algorithm 16.

Type restriction works well for both top-down (FOIL [25]) and bottom-up (GOLEM [90]) learners. In top-down systems such as FOIL [25] only clauses consistent with the type assignment are considered. FOIL constructs clauses by adding literals one by one to the most general clause. By checking that the introduced literals are consistent with the type assignment no clauses inconsistent with the type assignment are generated. Also, no consistent clause is excluded because of this greedy search heuristic. The reason for this is that any clause more specific then an inconsistent clause is also inconsistent.

In bottom up systems such as GOLEM [90], type assignment information is also of use. Once an *rlgg* that is consistent with the type information is formed any clause that is more general will also be consistent.

We would like to note that type meta knowledge is learned in the MOBAL [79] system. MOBAL generates type information by clustering constant terms which appear in the example set, this forms a lattice.

```
assign each predicate argument to a different type
assign each type to have an empty set of elements
foreach c ∈ E⁺ ∪ E⁻ ∪ B  do
  if c signals the need to merge  then
    merge types
  fi
  update type information
od
output  type information
```

Algorithm 16: Learning *type* properties

C.2.2 Functional dependency

Extraction of *mode* or functional dependency information proceeds as follows. First, the *merge* predicate is considered. Initially, it is assumed that each possible grouping of arguments in *merge* has functional dependency. This premise is then checked against the data. If the premise is false, then the data will show a counterexample, and the preprocessor will discard that possible functional dependency from its list of functional dependencies. More specifically, suppose it is being checked whether the first argument in *merge* is functionally dependent on the second argument (expressed as $Mode(merge(+, -, \ )))$. For this the *merge* predicate is scanned for any counterexamples. If a counterexample is found then such a functional dependency does not exist. In this case the first two facts provide a counterexample, as the first fact maps [] to [] and the second maps [] to [1]. No such conflict is found for $Mode(merge(+, +, -))$. By repeating this process for other predicates, *less*, *less-eq*, and *comp*, the remaining mode information may be discovered. For the *merge* example this information will be:

$$Mode(merge(+,+,-), merge(+,-,+), merge(-,+,+),$$
$$less(-,+), less(+,-), comp(+,-,-), comp(-,+,+)).$$

It should be noted that a functional dependency is induced in the *less* relation, even though no such dependency exists. This is due to the limited number of facts about *less* in the background knowledge. Algorithm 17 shows this inductive process.

Once again this information can be used to restrict the hypothesis search space. For example, $merge(X_1, X_2, X_3) \leftarrow comp(X1, X2, X3)$ is not consistent with the functional dependency information induced above.

```
foreach predicate name p ∈ E+ do
  foreach mapping of sets of argumaents to an argumant in p do †
    reset_table(T)
    foreach fact p(t1, t2, ..., tn) ∈ E+ do
      fun_input := input terms of p(t1, t2, ..., tn)
      fun_ouput := output term of p(t1, t2, ..., tn)
      if in_table(T, fun_input) then
        pre_output := get_value(T, fun_input)
        if fun_output ≠ pre_output then
          exit_loop†
        fi
      else
        insert_element(T, fun_input, fun_output)
      fi
    od
    output  functional dependence exists in p
  od
od
```

Algorithm 17: Learning functional dependencies

C.2.3 Symmetry

Meta knowledge about symmetry can also be derived from the data about the merge predicate. Algorithm 18 outlines the simple approach taken. Clearly, merge is symmetric in its first two arguments. The generation of this information involves testing each pair of arguments for symmetry. This test requires two steps. First, a scan of the merge ground facts is made to show there are no examples contrary to merge being symmetric in the pair of arguments under consideration. When the last two arguments are being tested for symmetry the positive example $merge([1], [], [1])$ and the negative example $merge([1], [1], [])$ signal that no symmetry exists. Second, some positive confirmation that the symmetry exists is found. When the first two arguments are being tested for symmetry the positive confirmation is first found in the examples *merge*$([],[1],[1])$ and *merge*$([1],[],[1])$.

It is easy to see that the information about symmetry can be used to restrict the hypotheses search space, as only one of the logically equivalent pair of clauses is considered. For example, the following two clauses are equivalent

$$\mathit{merge}(X_1, X_2, X_3) \leftarrow \mathit{comp}(X_1, X_4, X_5), \mathit{merge}(X_5, X_2, X_3).$$
$$\mathit{merge}(X_1, X_2, X_3) \leftarrow \mathit{comp}(X_1, X_4, X_5), \mathit{merge}(X_2, X_5, X_3).$$

because merge is symmetric in its first two arguments. So, there is no need to include both in the search space. Only MISST [55], LINUS [61], and MFOIL [34] use symmetry information.

```
foreach argument pair i, j in predicate p do
  flag := False
  foreach p(t_1, ..., t_i, ..., t_j, ..., t_n) ∈ E+ do
    if p(t_1, ..., t_j, ..., t_i, ..., t_n) ∈ E- then
      flag := True
    fi
  od
  if flag then
    output p is not symmetric in arguments i, j
  else
    output p is symmetric in arguments i, j
  fi
od
```

Algorithm 18: Learning symmetric properties

C.3 Experimental Results

The meta-knowledge extraction algorithms are a part of the LIME system. In this section we illustrate the effectiveness of these algorithms by incorporating them into a preprocessor that generates meta-knowledge for Quinlan's FOIL [25]. FOIL was chosen because it is a stable ILP system and it makes use of type information and key information[2]. Two series of experiments were carried out. First, nine target relations from Bratko's *PROLOG* book [21] were considered: *member*, *conc*, *del*, *sublist1*, *sublist2*, *permutation*, *add*, *gcd*, and *length*. These results, shown in Table C.3, contrast learning with generated meta-knowledge and learning with no meta-knowledge. The second set of experiments considered five of the examples that came with FOIL: *member*, *ncm*, *hinton*, *ackermann*, and *qs44* . These experiments, summarized in Table C.4, compare learning with preprocessor generated metaknowledge, learning with user supplied meta-knowledge, and learning without any metaknowledge. The experiments were carried out on a *DECstation 5000/33*. In the tables, '$\surd$' denotes that FOIL returned a correct hypothesis, '$\times$' denotes that FOIL returned an incorrect hypothesis, '∞' denotes that FOIL did not return a hypothesis.

[2]The key information, as will be seen later, can be 'derived' from the mode information.

Relation	FOIL Processing Times		Speed up	Preprocessing
	with learned m.k.[a]	no m.k.		
member	0.13$\surd$	0.20$\surd$	1.5	1.2
conc	0.87$\surd$	3.70$\surd$	4.6	3.4
del	0.22$\surd$	0.25$\surd$	1.1	1.5
sublist1	1.77$\times$	8.52$\surd$		13.5
sublist2	43.87$\surd$	47.98$\surd$	1.1	78.0
permutation	1.90$\surd$	8.65$\surd$	4.6	2.5
add	1.76$\surd$	1.77$\surd$	1.0	2.1
gcd	704.60$\surd$	718.32$\surd$	1.0	9.4
length	0.15$\surd$	2.60$\surd$	17.3	1.5

[a]meta-knowledge

Table C.3: Processing Times — Examples from Bratko's PROLOG book.

C.3.1 Examples from Bratko's book

Addition of preprocessor generated meta-knowledge to FOIL improved the performance of FOIL on seven out of nine relations as shown in Table C.3. On some relations the improvement in performance was significant. For one of the predicates, *sublist1*, addition of generated meta-knowledge actually caused FOIL to learn an incorrect hypothesis. The reason for this is explained below.

FOIL receives type information, specifying the types used, the elements that constitute the types, and the type of each argument in each relation. FOIL also receives a set of keys for each relation. Each key is a string where the characters map to the arguments of the relation. If the character is '`#`' then the corresponding argument must be bound; otherwise the character is '`-`' and the corresponding argument may be bound or unbound. FOIL only adds literals to the growing clauses that are consistent with a key binding information. If no keys are given for a relation, then any binding is valid. This information forms a semantic constraint on the clauses generated by FOIL. Information about keys may be induced from the functional dependency (mode) information learned from the raw data. When the preprocessor is used on a relation that contains functional dependencies, a set of *keys* is induced. However, this relation may be intended for use in a nondeterminate manner. This can be seen in the *sublist1* example; the clause to be induced is:

$$\mathit{sublist1}(A,B) \leftarrow \mathit{conc}(C,D,B), \mathit{conc}(A,E,D)$$

Since the *key* information for the concatenation relation, *conc*, is `##-/#-#/-##`, the above clause can not be induced by FOIL as C and D are not bound in the literal $\mathit{conc}(C,D,B)$. This is a significant problem, as the hypothesis search space has been over reduced, thereby excluding the correct hypothesis. One way this may be overcome is to only use key information as an initial restriction; removing it when no sensible logic program is generated. In *sublist2* example, the background knowledge includes both concatenation and composition (*conc* and *comp*) predicates, and this problem does not arise. Hence, *sublist2* can be induced with the generated meta-knowledge.

Relation	FOIL Execution Times					Pre-processing Time
	raw data, no m.k.[a]	with learned m.k.	m.k. explicitly provided[b]	*key* m.k. only	*type* m.k. only	
member	0.32√	0.18√	0.18√	0.30√	0.22√	3.8
ncm	263.93×	100.02√	18.48√	18.35√	272.83×	10.7
hinton	4.52√	5.32√	4.52√	4.52√	4.53√	4.1
ackermann	53.50√	50.92×	54.81√	55.03√	∞	3.6
qs44	∞	11771.53×	125.20√	155.30√	47332.80×	39.0

[a]meta knowledge

[b]These are the unmodified examples that come with FOIL.

Table C.4: Processing Times — FOIL's given examples.

C.3.2 Examples that came with FOIL

The experiments on the examples that came with FOIL turn out to be a bit more tricky (Table C.4). For all these examples, the preprocessor is successful in inducing the intended *type* meta-knowledge. Also, it is able to induce a super set of the *key* meta-knowledge. One would expect that if FOIL could learn an example without any *key* meta-knowledge, then FOIL would be able to learn the example, perhaps even faster, with a super set of the correct *key* meta-knowledge. Unfortunately, there are problems with this expectation. This can be seen in the context of the *ackermann* example, where correct meta-knowledge prevents learning. The extra information misdirects FOIL's greedy search technique. In the *member* and *ncm* examples, using generated meta-knowledge improves the performance of FOIL compared to not using meta-knowledge. In fact, *ncm* can not be learned without this meta-knowledge. In *hinton*, introducing meta-knowledge has a slightly negative effect. FOIL could not learn *qs44* with generated meta-knowledge, as it does not create a sufficient restriction on the hypothesis search space. It should be noted that FOIL is able to learn *qs44* with user supplied meta-knowledge, but that is because after trial and error a user can just give the right bit of meta-knowledge that does not throw FOIL off. Hence, in such situations, the preprocessor can be used as a guide to the user in selecting the right bit of meta-knowledge.

The *ncm* (n-choose-m) example is interesting. First, only integers are used so introducing *type* meta-knowledge has no effect. Second, *key* information is necessary for FOIL to induce the correct clauses, when this *type* information is induced by the preprocessor, FOIL can induce the correct hypothesis.

Third, the *key* information for the *decrement* predicate is the only difference between the data with user supplied meta-knowledge and data with preprocessor generated meta-knowledge. The unmodified data only allows *dec* to be used as the decrement function:

```
*dec(M,M) #-
```

Whereas when this meta-knowledge is induced by the preprocessor, *dec* may be used either as decrement function: or as increment function

```
*dec(M,M) #-/-#.
```

It should be noted how this small change has a significant impact on the hypothesis search space, and hence, a significant effect in the performance of the ILP system.

In the case of *ackermann*, *key* information misdirects the greedy search algorithm. Hence, FOIL is unable to learn a correct hypothesis, even though the *key* information is consistent with the target hypothesis. Notice that the 'raw data no m.k.' and 'm.k. explicitly provided' input files are identical, as there is no *key* information in the *ackermenn* example provided with FOIL.

The preprocessor was implemented in the functional programming language *Gofer*. When it is re-implemented in *C* and included within FOIL, its effect on the overall system performance would be insignificant. So, preprocessing time is not included in the calculation of speed-up.

C.4 Theoretical Properties of the Algorithms

An informal summary of the theoretical properties of Algorithms 16, 17, and 18 are presented in this section. Let n be the number of tuples (that is the size of $\mathcal{E}^+ \cup \mathcal{E}^- \cup \mathcal{B}$) and let k be the number of arguments in the predicate with the maximum number of arguments. In practice, k is fairly small.

The time complexity of the *type* generation algorithm is $O(n \cdot k)$. There is no guarantee the algorithm will find the intended *type* meta-knowledge—not even in the limit. There are two problems: first, there may be no examples to deduce arguments having the same type; second, the intended type assignment may have overlapping elements. However, this is not a problem when the type assignment is only being used to restrict the hypothesis search space. The first over-restricts the hypothesis search space, but the hypotheses it removes could not cover any positive examples, so it is worth removing them regardless. The second under restricts the hypothesis search space, to only cause a reduction in the performance of the ILP system.

The time complexity of the *functional dependency* generation algorithm is $O(n \cdot 2^k)$ and the time complexity of the *symmetry* generation algorithm is $O(n \cdot k^2)$. Both the *functional dependency* and *symmetry* generation algorithms deduce the correct meta-knowledge when the closed world assumption is used. However, if the closed world assumption is not used, then neither of the algorithms can guarantee inducing the intended result. Although, both of the algorithms learn the intended meta-knowledge in the limit.

C.5 Discussion

The preprocessor described in this appendix has successfully demonstrated the usefulness of extracting three kinds of meta-knowledge from raw data. The capability of the preprocessor is being expanded to further extract anti-symmetric, exclusive, transitive, and reflexive properties from relational data.

The development of algorithms for extracting various kinds of meta-knowledge is aimed at the design of an incremental ILP system that dynamically changes its hypothesis generation strategy based on the feedback provided from the meta-knowledge discovered. We illustrate this with the help of an example. Suppose, an ILP system is learning a predicate, and it begins with a conservative hypothesis generation heuristic by assuming that the predicate is symmetric with respect to certain arguments. In case, its assumption is true, the system continues to use this heuristic. However, if a new fact provides a counterexample to the symmetry assumption, the system widens its hypothesis generation heuristic. Such a system can be seen to make a more prudent search of the hypothesis space.