Wrapper Induction for Information Extraction

by

Nicholas Kushmerick

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

1997

Approved by____

(Chairperson of Supervisory Committee)

Program Authorized
to Offer Degree____

Date____

University of Washington

Abstract

Wrapper Induction for Information Extraction

by Nicholas Kushmerick

Chairperson of Supervisory Committee: *Professor Daniel S. Weld*
Department of Computer Science
and Engineering

The Internet presents numerous sources of useful information—telephone directories, product catalogs, stock quotes, weather forecasts, *etc.* Recently, many systems have been built that automatically gather and manipulate such information on a user's behalf. However, these resources are usually formatted for use by people (*e.g.*, the relevant content is embedded in HTML pages), so extracting their content is difficult.

Wrappers are often used for this purpose. A wrapper is a procedure for extracting a particular resource's content. Unfortunately, hand-coding wrappers is tedious. We introduce *wrapper induction*, a technique for automatically constructing wrappers. Our techniques can be described in terms of three main contributions.

First, we pose the problem of wrapper construction as one of *inductive learning*. Our algorithm learns a resource's wrapper by reasoning about a sample of the resource's pages. In our formulation of the learning problem, *instances* correspond to the resource's pages, a page's *label* corresponds to its relevant content, and *hypotheses* correspond to wrappers.

Second, we identify several classes of wrappers which are *reasonably useful, yet efficiently learnable*. To assess usefulness, we measured the fraction of Internet resources that can be handled by our techniques. We find that our system can learn

wrappers for 70% of the surveyed sites. Learnability is assessed by the asymptotic complexity of our system's running time; most of our wrapper classes can be learned in time that grows as a small-degree polynomial.

Third, we describe *noise-tolerant techniques for automatically labeling the examples*. Our system takes as input a library of *recognizers*, domain-specific heuristics for identifying a page's content. We have developed an algorithm for automatically *corroborating* the recognizer's evidence. Our algorithm perform well, even when the recognizers exhibit high levels of noise.

Our learning algorithm has been fully implemented. We have evaluated our system both analytically (with the PAC learning model) and empirically. Our system requires 2 to 44 examples for effective learning, and takes about ten seconds of CPU time for most sites. We conclude that wrapper induction is a feasible solution to the scaling problems inherent in the use of wrappers by information-integration systems.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

First of all, I thank my family: without your inspiration and support, none of this would have been possible. Thanks for not asking too many questions, or too few.

The fabulous Marilyn McCune demands special mention. Marilyn did not merely put up with far too much during the past eight months. Her wit and confidence in the face of my gloom, her affection in the face of my six o'clock mornings, her verse and laughter in the face of my equations, and her culinary ingenuity in the face of my hunger—in these and a thousand other ways, Marilyn carried the day.

Simply put, Dan Weld has been a fantastic advisor. I am grateful for the boundless faith in me he showed, even as I insisted on exploring yet another patently ludicrous *cul-de-sac*. And though it was sometimes intimidating, Dan's comprehensive grasp of the several areas in which we worked was always a helpful source of new ideas.

My colleagues at the University of Washington have provided a fertile environment for exploring artificial intelligence and computer science, not to mention life. Let me thank in particular Tony Barrett, Paul Barton-Davis, Adam Carlson, Bob Doorenbos, Denise Draper, Oren Etzioni, Marc Friedman, Keith Golden, Steve Hanks, Anna Karlin, Neal Lesh, Mike Perkowitz, Ted Romer, Eric Selberg, Stephen Soderland and Mike Williamson.

Brett Grace provided invaluable assistance in transforming my crude vision into the WIEN application. Finally, Boris Bak's assistance with the "search.com" survey is much appreciated.

This dissertation is dedicated with love to my grandmothers, Mary Nowicki Knoll and Marie Naglak Kushmerick, who got everything started.

Chapter 1

INTRODUCTION

1.1 Motivation

1.1.1 Background: Information resources and software agents

The much-heralded "information technology age" has delivered a stunning variety of on-line information resources: telephone directories, airline schedules, retail product catalogs, weather forecasts, stock market quotations, job listings, event schedules, scientific data repositories, recipe collections, and many more. With widespread adoption of open standards such as HTTP, and extensive distribution of inexpensive software such as Netscape Navigator, these resources are becoming ever more widely available.

As originally envisioned, this infrastructure was intended for use directly by *people*. For example, Internet resources often use query mechanisms (*e.g.*, HTML forms) and output standards (*e.g.*, HTML's formatting constructs) that are reasonably well suited to direct manual interaction.

An alternative to manual manipulation is *automatic* manipulation: the use of computer programs (rather than people) to interact with information resources. For many, the information feast has become an information glut. There is a widely-recognized need for systems that automate the process of managing, collating, collecting, finding, filtering, and redistributing information from the many resources that are available.

Over the last several years, the artificial intelligence community has responded to this challenge by developing systems that are loosely allied under the term *software agents*. In this thesis, we are mainly motivated by a specific variety of such agents: those which use an array of existing information resources as *tools*, much as a house-cleaning robot might use vacuum cleaners and mops. The idea is that the user specifies *what* is to be accomplished; the system figures out *how* to use its tools to accomplish the desired task. Following the University of Washington terminology, we will call such systems *softbots* (<u>soft</u>ware ro<u>bots</u>).[1]

To make this discussion concrete, we will focus on one particular system. The RAZOR (née OCCAM) system [Kwok & Weld 96, Friedman & Weld 97] accepts queries that describe a particular information need (*e.g.*, "Find reviews of movies showing this week in Seattle by Fellini"). RAZOR then computes and executes a sequence of information-gathering actions that will satisfy the query. In the example, the actions might involve: querying various movie-information sites (*e.g.*, "imdb.com") to obtain a list of Fellini's movies; then, asking theaters' sites (*e.g.*, "showtimes.hollywood.com") which of these movies are now showing; and finally, passing these movies on to various sites containing reviews (*e.g.*, "siskel-ebert.com").

Softbots are attractive because they promise to relieve users of the tedium of manually carrying out such operations. For example, for each of the three steps above, there are many potentially relevant resources. Moreover, information gleaned at each step must be manually passed on (*i.e.*, read from the browser and typed into an HTML form) to each of the resources consulted in the next step.

[1] The software agent literature is vast; see [Etzioni et al. 94, Wooldridge & Jennings 95, Bradshaw 97] or [www.agents.org] for surveys. The softbot paradigm is discussed in detail in [Etzioni & Weld 94]. Examples of the systems we have in mind include [Etzioni et al. 93, Chawathe et al. 94, Kirk et al. 95, Carey et al. 95, Krulwich 96, Kwok & Weld 96, Arens et al. 96, Shakes et al. 97, Doorenbos et al. 97, Selberg & Etzioni 97, Decker et al. 97], as well as commercial products such as Jango [www.jango.com], Junglee [www.junglee.com], Computer ESP [oracle.-uvision.com/shop], and AlphaCONNECT [www.alphamicro.com]. This short list certainly neglects many important projects. In Chapter 8 we discuss related work in detail.

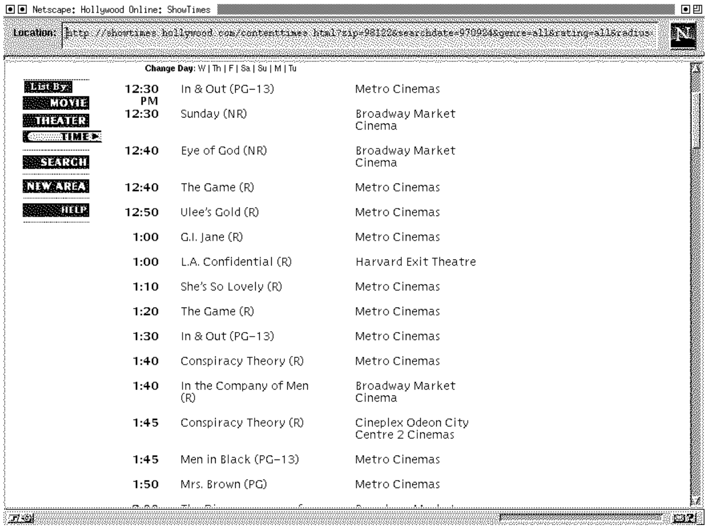
One might argue that this idea—the automatic manipulation of resources intended for people—is somewhat misguided. Haven't the distributed databases and software agents communities developed elaborate protocols for exchanging information across heterogeneous environments?[2] Why not re-engineer the information resources so that they provide interfaces that are more conducive to automatic interaction?

This is a reasonable objection. However, organizations might have many reasons for not wanting to open up their information resources to software agents. An on-line business might prefer to be visited manually rather than mechanically. Search engines such as Yahoo!, for example, are in the business of delivering users to advertisers, *not* servicing queries as an end in itself. Similarly, a retail store might not want to simplify the process of automatically comparing prices between vendors. And of course the cost of re-engineering existing resources might be prohibitive.

For these reasons, if we want to build software agents that access a wide variety of information resources, the only option might be to build systems that make use of the existing interfaces that were intended originally for use by people. At the highest level, this challenge—**building systems that can use human-centered interfaces**—constitutes the core motivation of this thesis.

Of course, this challenge is exceedingly difficult. For example, machines today can't fully understand the unrestricted natural language text used in on-line newspaper or magazine articles. Thus an entirely general-purpose solution to this challenge is a long way off. Instead, following standard practice, our approach is to isolate a relevant yet realistic special case of this difficult problem.

[2] There are many such proposals, each developed under somewhat different motivations. A representative sample includes CORBA [www.omg.org], ODBC [www.microsoft.com/data/odbc], XML [www.w3.org/TR/WD-xml], KIF [logic.stanford.edu/kif], Z39.50, [lcweb.loc.gov/z3950], WIDL [www.webmethods.com], SHOE [Luke et al. 97], KQML [Finin et al. 94] and the Metacontent Format [mcf.research.apple.com].



http://showtimes.hollywood.com/contenttimes.html?zip=98122&searchdate=970924&genre=all&rating=all&radius=

Change Day: W | Th | F | Sa | Su | M | Tu

12:30 PM	In & Out (PG-13)	Metro Cinemas
12:30	Sunday (NR)	Broadway Market Cinema
12:40	Eye of God (NR)	Broadway Market Cinema
12:40	The Game (R)	Metro Cinemas
12:50	Ulee's Gold (R)	Metro Cinemas
1:00	G.I. Jane (R)	Metro Cinemas
1:00	L.A. Confidential (R)	Harvard Exit Theatre
1:10	She's So Lovely (R)	Metro Cinemas
1:20	The Game (R)	Metro Cinemas
1:30	In & Out (PG-13)	Metro Cinemas
1:40	Conspiracy Theory (R)	Metro Cinemas
1:40	In the Company of Men (R)	Broadway Market Cinema
1:45	Conspiracy Theory (R)	Cineplex Odeon City Centre 2 Cinemas
1:45	Men in Black (PG-13)	Metro Cinemas
1:50	Mrs. Brown (PG)	Metro Cinemas

$\Downarrow$ information extraction

$$\left\{\begin{array}{c}\langle\texttt{12:30},\ \texttt{In \& Out},\ \texttt{PG-13},\ \texttt{Metro Cinemas}\rangle,\\ \langle\texttt{12:30},\ \texttt{Sunday},\ \texttt{NR},\ \texttt{Broadway Market Cinema}\rangle,\\ \langle\texttt{12:40},\ \texttt{Eye of God},\ \texttt{NR},\ \texttt{Broadway Market Cinema}\rangle,\\ \vdots\end{array}\right\}$$

Figure 1.1: *The "showtimes.hollywood.com" Internet site is an example of a semi-structured information resource.*

1.1.2 Our focus: Semi-structured resources

Fortunately, many of the information resources we want our software agents to use do not exhibit the full potential complexity suggested by the challenge just outlined. For instance, Figure 1.1 illustrates that the resource "showtimes.hollywood.com" does not contain unrestricted natural language text. Rather, it presents information in a highly regular and structured fashion.

Specifically, "showtimes.hollywood.com" structures its output in the form of a table. The table contains four columns (time, movie, rating and theater), and one row for each quadruple of information in the document. Now, if we are to build a software agent that can make use of this resource, then we must provide it with a procedure which, when invoked on such a document, extracts the document's content. In the literature, such specialized procedures are commonly called *wrappers* [Papakonstantinou et al. 95, Chidlovskii et al. 97, Roth & Schwartz 97]

This thesis is concerned with such *semi-structured* information resources—those that exhibit regularity of this nature. This focus in motivated by three concerns:

- Semi-structured resources generally do not employ unrestricted natural language text, but rather exhibit a fair degree of structure. Thus we are optimistic that handling semi-structured resources is a *realistic* goal.
- On the other hand, semi-structured resources contain extraneous elements that must be ignored, such as advertisements and HTML formatting constructs. Most importantly, there is no machine-readable standard explaining how to interact with this site, or how it renders information. Thus extraction from semi-structured documents is *not entirely trivial*.
- As discussed in [Doorenbos et al. 97, Perkowitz et al. 97], we hope that a relatively large fraction of actual Internet information resources are semi-structured in this way, so that our results are *reasonably useful*.

Besides semi-structured resources, the discussion so far has hinted at our second main focus: *information extraction*. Roughly, by the phrase "information extraction" we refer to the process of identifying and organizing relevant fragments in a document while discarding extraneous text.[3] Once a resource's information is extracted in this manner, it can be manipulated in many different ways. Thus while we are strongly

[3] Of course, this emphasis of extraction ignores many important issues, such as how software agents come to *discover* resources in the first place [Bowman et al. 94, Zaiane & Jiawei 95], how they can learn to *query* these resources [Doorenbos et al. 97], and which services beyond extraction wrappers should provide [Papakonstantinou et al. 95, Roth & Schwartz 97]. Also, while the essence is similar, our use of the phrase *information extraction* differs from other uses [Cowie & Lehnert 96]. In Chapter 8, we describe this and other related work in detail.

motivated by the software agent paradigm, this thesis is not concerned with specific systems or architectures. Rather, we have developed an enabling technology—namely, techniques for automatically constructing wrappers—that we expect to be relevant to a wide spectrum of software agent work.

1.1.3 An imperfect strategy: Hand-coded wrappers

To summarize, we have introduced the notion of semi-structured information resources. The hope is that we can construct relatively simple information extraction procedures (which we'll call wrappers) for semi-structured resources.

For example, it turns out that documents from "showtimes.hollywood.com" can be parsed using an extremely simple mechanism. For now, we omit all the details, but the basic idea is that the text fragments to be extracted (show-time, movie title, *etc.*) are always surrounded by certain specific strings. Note, for example, that the show-times are consistently rendered in bold face; inspection of the HTML document reveals that each show-time is surrounded by the tags `<B>`···`</B>`.

This observation leads to a very simple parsing procedure: the two strings `<B>` and `</B>` can be used to identify and extract the relevant fragments of the document. To be sure, this simple mechanism can't handle some important subtleties. For now, the point is simply that a very simple mechanism can perform this sort of information extraction task, since the wrappers can exploit fortuitous regularities in a collection of semi-structured documents.

Given that such wrappers exists, the question becomes: how should designers of software agents build them? One obvious possibility is to construct an agent's wrappers *by hand*; indeed, nearly all wrappers today are constructed by hand.

Unfortunately, while straightforward in principle, hand-coding wrappers is time-consuming and error-prone, mainly because accurately determining the appropriate delimiter strings (*e.g.* `<B>` and `</B>`) is tedious. Note also that a specialized wrapper must be written for *each* of the resources in a software agent's arsenal. Moreover, actual Internet sites change their format occasionally, and each such modification might require that the wrapper be rewritten. To summarize, hand-coding results in a serious knowledge-engineering bottleneck, and software agents that rely on hand-coded wrappers face serious scaling problems.

Set against this background, our primary motivation can now be stated quite succinctly. **To alleviate this engineering bottleneck, we seek to automate the process of constructing wrappers for semi-structured resources.**

In the remainder of this chapter, we first provide an overview of our approach to automating the wrapper construction process (Section 1.2). We then summarize our major contributions (Section 1.3), and describe how the chapters of this thesis are related and organized (Section 1.4).

1.2 Overview

1.2.1 Our solution: Automatic wrapper construction

Our goal is to automatically construct wrappers. Since a wrapper is simply a computer program, we are essentially trying to do automatic programming. Of course, in general automatic programming is very difficult. So, as suggested earlier, we follow standard practice and proceed by isolating particular classes of programs for which effective automatic techniques can be developed.

For example, the `<B>`···`</B>` technique introduced earlier for extracting the showtimes from “showtimes.hollywood.com” suggests one simple class of wrappers, which we call LR (left-right). LR wrappers operate by scanning a document for such delimiters, one pair for each attribute to be extracted. An LR wrapper starts by scanning forward to the first occurrence of the left-hand delimiter for the first attribute, then to the right-hand delimiter for the first attribute, then to the left-hand delimiter for the second attribute, then to the right-hand delimiter for the second attribute, and so forth, until all the “columns” have been extracted from the table’s first “row”.

The wrapper then starts over again with the next row; execution halts when all rows have been extracted.

Admittedly, LR is an extremely simple class of wrappers. Can suitable delimiters be found for real information resources? According to our recent survey of actual Internet resources, LR wrappers are sufficiently expressive to handle 53% of the kinds of resources to which we would like our software agents to have access (see Section 7.2 for details). Moreover, as we discuss next, we have developed efficient algorithms for automatically coding LR wrappers. The LR class, therefore, represents a reasonably useful strategy for alleviating the wrapper construction bottleneck.

While LR is reasonably expressive, by inspecting the 47% of resources that it can not handle, we have developed a more general wrapper class, which we call HLRT (head–left–right–tail). HLRT wrappers work like LR wrappers, except that they ignore distracting material in a document's head (its top-most portion) and tail (bottom).

HLRT is more complicated than LR, but this additional machinery pays modest dividends: about 57% of the resources we surveyed can be handled by HLRT. Moreover, this extra expressiveness costs little in terms of the complexity of automatic wrapper construction: just as with LR, we have developed efficient algorithms for automatically coding HLRT wrappers.

Since HLRT is more complicated than LR, it exposes more of the subtleties of automatic construction. Therefore, in this thesis we focus mainly on HLRT; see Chapter 4. However, in Chapter 5, we describe LR as well as four other wrapper classes. All six classes are based on the idea of using delimiters such as `<B>···</B>` to extract the attributes. The classes differ in two ways. First, we developed various techniques to avoid getting confused by distractions (*e.g.*, advertisements). Second, we developed wrapper classes for extracting information that is laid out not as a table, but rather as a hierarchically nested structure (*e.g.*, a book's table of contents).

The main result of this analysis is a comparison of the six wrapper classes on two bases: *relative expressiveness*, a measure of the extent to which one wrapper class can "mimic" another; and the *computational complexity* of automatically constructing wrappers in each class.

1.2.2 Our technique: Inductive learning

We have developed techniques to automatically construct various types of wrappers. How does our system work? Our approach is based on inductive learning, a well-studied paradigm in machine learning; see, for example, [Dietterich & Michalski 83, Michalski 83] and [Mitchell 97, Chapters 2–3]. Induction is the process of reasoning from a set of *examples* to an *hypothesis* that (in some application-specific sense) generalizes or explains the examples. A inductive learning algorithm, then, takes as input a set of examples, and produces as output an hypothesis. For example, if told that '*Thai food is spicy*', '*Korean food is spicy*' and '*German food is not spicy*', an inductive learner might output '*Asian food is spicy*'.

Induction is, in principle, very difficult. The fundamental problem is that *many* hypotheses are typically consistent with a set of examples, but the learner has no basis on which to choose. For example, while we might judge '*Asian food is spicy*' as a reasonable generalization, on what basis do we reject the trivial generalization '*Thai or Korean food is spicy*', formed by simply disjoining the examples? The standard practice in machine learning is to *bias* [Mitchell 80] the learning algorithm so that it considers only hypothesis that meet certain criteria. For example, the learning algorithm could be biased so that is does not consider disjunctive hypotheses.

Wrapper induction. This thesis was strongly influenced by two related University of Washington projects: ILA [Perkowitz & Etzioni 95] and SHOPBOT [Doorenbos et al. 97]; see also [Etzioni 96b, Perkowitz et al. 97]. This seminal work proposed a framework in which softbots use machine learning techniques to learn about the tools they use. The idea is that during an off-line learning phase, the agents interact with these tools, generalizing from the observed behavior. The results of this learning phase are then used on-line to satisfy users' queries.

We can summarize this framework in terms of the following observation: **an effective way to learn about an information resource is to reason about a sample of its behavior.** Our work can be posed in terms of this observation as follows: we are interested in the following *wrapper induction problem*

input: the examples correspond to samples of the input–output behavior of the wrapper to be constructed;

output: a hypothesis corresponds to a wrapper, and hypothesis biases correspond to classes of wrappers, such as LR and HLRT.

Under this formulation, an effective wrapper induction system is one that rapidly computes a wrapper that behaves as determined by the input–output sample.

Like all induction algorithms, our system assumes that there exists some *target* wrapper which works correctly for the information resource under consideration. The input to our system is simply a sample of this target wrapper's input–output behavior. In the "showtimes.hollywood.com" example, this sample might include the input–output pair shown in Figure 1.1.

The desired output for "showtimes.hollywood.com" is the target wrapper. How does our learning algorithm reconstruct the target from the input–output samples it takes as input? The basic idea is that our system considers the set of *all* wrappers, rejecting those that are inconsistent with the observed input–output samples.

Initially, any wrapper is potentially consistent. Then, by examining the examples (such as Figure 1.1), the system eliminates incompatible wrappers. Guided by the sample input–output behavior of the target, our system examines the text extracted from the example documents in order to find incompatibilities. For instance, when the system observes that the extracted show-times are always surrounded by `<B>`···`</B>`, it can discard all wrappers that do not use these delimiters.

Of course, the set of all wrappers is enormous; in fact it is infinite. Thus a key to effective wrapper learning is to reason efficiently over these sets. As described in

Chapters 4 and 5, we have built and analyzed the efficiency of learning algorithms for six different wrapper classes.

Automatically generating the examples. As the wrapper induction problem was stated, our learning algorithm takes as input a sample of the behavior of the very wrapper to be learned. Although this assumption is standard in the inductive learning community, we must ask whether it is appropriate for our application.

First, note that the apparent contradiction—to learn a wrapper, we have to first sample its behavior—is easily resolved. The input to our learning system need only be a sample of how the target wrapper *would* behave, *if* it were given an example document from the resource under consideration.

This observation suggests one simple way to gather the required input: ask a person. Under this approach, we have reduced the task of hand-coding a wrapper to the task of hand-labeling a set of example documents. In some cases, this reduction may be quite effective at simplifying the person's cognitive load. For instance, less expertise might be required, since the user can focus on the attributes to be extracted (show-times, movie titles, *etc.*), rather than on "implementation details" (verifying that `<B>`$\cdots$`</B>` is a satisfactory pair of delimiters, *etc.*).

Nevertheless, we seek to automate wrapper construction as much as possible. To that end, we have developed a set of techniques for automatically labeling example documents; see Chapter 6. Our labeling algorithm takes as input domain-specific heuristics for recognizing instances of the attributes to be extracted. In the "show-times.hollywood.com" example, our labeling system would take as input a procedure for recognizing all of the instances of times (text fragments such as `12:30`, `2:50` and `9:07`).

The required recognition heuristics might be very primitive—*e.g.*, using the regular expression `1?[0-9]:[0-9][0-9]` to identify show-times. At the other extreme, recognition might require natural language processing, or the querying of other information resources—*e.g.*, asking an already-wrapped resource to determine whether a particular text fragment is a movie title. While such recognition heuristics are certainly very important, this thesis is not concerned with either the theory or practice of developing these heuristics. Rather, our system simply requires that these heuristics be provided as input, and then treats them as "black boxes" when determining a resource's structure.

Once the instances of each attribute have been identified, our labeling system combines the results for the entire page. If the recognition knowledge is perfect, then this integration step is trivial. However, note that perfect recognition heuristics do not obviate the need for wrapper induction, because while the heuristic might be perfect, they might also be very slow and thus be unable to deliver the fast performance demanded by a software agent's on-line information extraction subsystem.

An important feature of our system is that it can tolerate quite high rates of "noise" in the recognizer heuristics. For example, the "time" regular expression above might find some text fragments that are not in fact show-times, or it might mistakenly ignore some of a document's show-times. Our automatic page labeling algorithm can make use of recognizers even when they make many such mistakes; for example, we have tested our system using recognizer heuristics that are wrong up to 40% of the time, and found only a modest performance degradation (see Section 7.3). The intuition for this capability is that while the recognition heuristics might fail for any *particular* show-time, they are unlikely to fail repeatedly across *several* example documents.

How many examples are enough? We presented a framework, inductive learning, with which to understand our approach to automatic wrapper construction. We then went on to describe our technique for solving one problematic aspect of this approach: the need for samples from the very wrapper to be learned. The basic inductive learning model also has little to say regarding a second important issue:

how many examples must our learning system examine in order to be confident that it will output a satisfactory wrapper?

Computational learning theory, a subfield of the machine learning and theoretical computer science communities, provides a rigorous foundation for investigating the expected performance of learning systems; see [Angluin 92] for a survey. We have applied these techniques to our wrapper induction application.

Our results (see Section 4.6) are statistical in nature. We have developed a model which predicts how many examples our learning algorithm must observe to ensure that, with high probability, the algorithm's output wrapper makes a mistake (*i.e.*, fails when parsing an unseen document) only rarely. This investigation is formalized in terms of user-specified parameters that define "high probability" and "rarely". Based on the structure of the wrapper learning task, we use these parameters to derive a bound on the number of examples needed to satisfy the satisfy the stated criterion. Thus our wrapper induction system usually outputs a wrapper that rarely fails; in computational learning theory terminology, our system outputs wrappers that are *probably approximately correct* (PAC).[4]

1.2.3 Evaluation

Perhaps the most important issues in any research project concern evaluation: how can one know the extent to which one's results are relevant and important? In Chapter 7, we take a thoroughly empirical approach to evaluation, measuring the behavior of our system against actual information resources on the Internet.

Our first experiment takes the form of a survey (Section 7.2). We are interested in whether the wrapper classes we have developed are useful for handling actual Internet resources. We examined a large pool of such resources, and determined for each whether it could be handled by one or more of the six wrapper classes

[4] The PAC model was proposed first in [Valiant 84]; see [Martin & Biggs 92, Kearns & Vazirani 94] for extensive treatments.

developed in this thesis. The results are quite encouraging: 53% of the resources can be handled by the LR wrapper class, while the HLRT class can handle 57%; the remaining four wrapper classes have a similar degree of expressiveness. We conclude that we have identified wrapper classes that are useful for numerous interesting, real-world information resources.

A second set of experiments tests whether our induction algorithm is too expensive to be used in practice (Section 7.3). One important resource is time: averaged across several real domains, our system requires about one minute to learn an HLRT wrapper.

While CPU time is an important measure, recall from our discussion of the PAC model that another important resource is the required set of examples. Each example document must be fetched from the resource, and then labeled (either by a person or with our automatic labeling techniques).

Given these costs, we would prefer that our induction system use few examples. We find that, averaging across several actual domains, our system requires 2–44 examples in order to learn a wrapper that works perfectly against a large suite of test documents. As a point of comparison, we can relate this result with our theoretical PAC bound. Our model predicts that 300–1100 examples are required. Thus our PAC bound is too loose by one to two orders of magnitude; improving this bound is a challenging direction for future research; see Chapter 9.

Finally, we have developed the wrapper induction environment (WIEN) application (Section 7.8). Using a standard Internet browser, a user shows WIEN an example document, and then uses the mouse to indicate the portions of the page to be extracted. In addition, the user can supply recognizer heuristics which are automatically applied to the document. WIEN then tries to learn a wrapper for the resource. When shown a second example, WIEN uses the learned wrapper to automatically label the new example. The user then corrects any mistakes, and WIEN generalizes from both examples. This process repeats until the user is satisfied.

1.3 Contributions

As discussed earlier, the primary motivation of this thesis involves enabling software agents to interact with the human-centric interfaces found at most Internet information resources. Since this is a very hard problem, we have focused on a more modest challenge: the development of techniques for automatically constructing wrappers for semi-structured resources.

Let us summarize this chapter by explicitly stating what we believe to be our major contributions towards meeting this challenge.

1. We crisply pose the automatic wrapper construction problem as one of inductive (*i.e.*, example-driven) learning (Chapters 2 and 3).
2. We identify several classes of wrappers which are expressive enough to handle numerous actual Internet resources, and we develop efficient algorithms for automatically learning these classes (Chapters 4 and 5).
3. Finally, we develop a method for automatically labeling the examples required by our induction algorithm. An important feature of our method is that it is robust in the face of noise (Chapters 6).

1.4 Organization

The remainder of this thesis is organized as follows.

Chapter 2: A formal model of information extraction.
We begin by stating the kind of information extraction tasks that concern us.

Chapter 3: Wrapper construction as inductive learning.
We then show how to frame the problem of automatically constructing wrappers as one of inductive learning.

Chapter 4: The HLRT wrapper class.
We then illustrate our approach with one particular wrapper class, HLRT; our results include an efficient and formally correct learning algorithm, and a PAC-theoretic analysis of the number of training

examples needed to learn a satisfactory wrapper.

Chapter 5: Beyond HLRT: Alternative wrapper classes.

HLRT is but one of many wrapper classes; in this chapter we explore five more, including classes for extracting information from hierarchically nested (rather than tabular) documents.

Chapter 6: Corroboration.

The learning algorithm developed in Chapter 4 requires a set of labeled examples as input. In Chapter 6 we describe techniques for automating this labeling process.

Chapter 7: Empirical evaluation.

We describe several experiments which demonstrate the feasibility of our approach on actual Internet information resources, and describe WIEN, an application which embodies many of the ideas developed in this thesis.

Chapter 8: Related work.

Our work is motivated by and draws insights from a variety of different research areas.

Chapter 9: Future work and conclusions.

We suggest avenues for future research, and summarize our contributions and conclusions.

Appendix A: An example resource and its wrappers.

We provide the complete HTML source from an example Internet information resource, and show a wrapper for this resource for each of the wrapper classes described in Chapter 5.

Appendix B: Proofs.

We include proofs of the theorems and lemmas asserted in this thesis.

Appendix C: String algebra.

Finally, we provide details of the character string notation used throughout this thesis.

Enjoy!

Chapter 2

A FORMAL MODEL OF INFORMATION EXTRACTION

2.1 Introduction

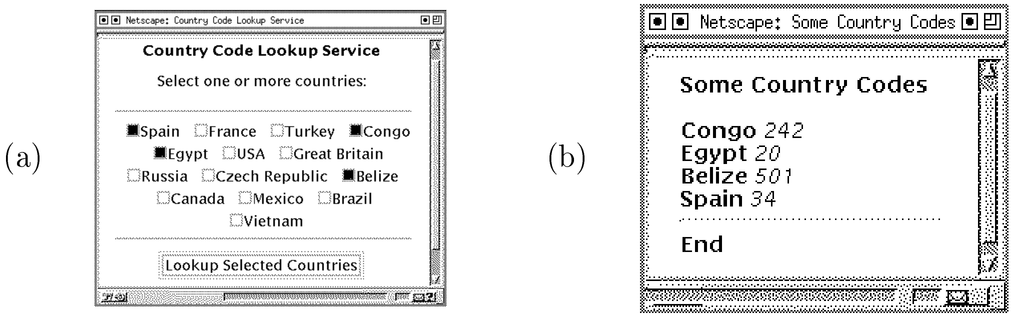
This thesis is concerned with learning to extract information from semi-structured information resources such as Internet sites. In this chapter, we temporarily ignore issues related to learning, and develop a formal model of the extraction process itself. This preliminary step is important, because it allows us to state precisely what we want our system to learn.

We start with a high-level overview of our model (Section 2.2). We then formalize and provide a precise notation for these intuitions and ideas (Section 2.3).

2.2 The basic idea

Figure 2.1 provides an example of the sort of information resource with which we are concerned. The figure shows a fictitious Internet site that provides information about countries and their telephone country codes. When the form shown in 2.1(a) is submitted, the resource responds as shown in 2.1(b), which was rendered from the HTML document shown in 2.1(c).

More generally, an *information resource* is a system that responds to *queries*, yielding a *query response* that is (presumably) appropriate to the query. Usually, the resource contains a database; incoming queries are posed to the database, and the results are used to compose the response. However, from the perspective of this thesis, the information resource is simply a black box, and we will have very little to say about either the queries or the resource's internal architecture.


(a)
Netscape: Country Code Lookup Service
(b)
Netscape: Some Country Codes

(a)

Country Code Lookup Service

Select one or more countries:

■Spain □France □Turkey ■Congo
■Egypt □USA □Great Britain
□Russia □Czech Republic ■Belize
□Canada □Mexico □Brazil
□Vietnam

Lookup Selected Countries

(b)

Some Country Codes

Congo *242*
Egypt *20*
Belize *501*
Spain *34*

End

(c)

```
<HTML><TITLE>Some Country Codes</TITLE><BODY>
<B>Some Country Codes</B><P>
<B>Congo</B> <I>242</I><BR>
<B>Egypt</B> <I>20</I><BR>
<B>Belize</B> <I>501</I><BR>
<B>Spain</B> <I>34</I><BR>
<HR><B>End</B></BODY></HTML>
```

Figure 2.1: (a) *A fictitious Internet site providing information about countries and their telephone country codes;* (b) *an example query response; and* (c) *the* HTML *text from which (b) was rendered.*

In the country/code example, the response is presented as an Internet *page* (to use the standard jargon). We intend that the results of this thesis apply more broadly than just to HTML documents on the Internet. Nevertheless, for the sake of simplicity, we will use the terms *page* and *query response* interchangeably.

The example information resource provides information about two *attributes*: the names of the countries, and their telephone country codes. In this thesis, we adopt a standard relational data model: query responses are treated as providing one or more *tuples* of relevant information. Thus, the example response page provides four $\langle$country, code$\rangle$ pairs:

$$\{\langle \texttt{Congo, 242}\rangle,\ \langle \texttt{Egypt, 20}\rangle,\ \langle \texttt{Belize, 501}\rangle,\ \langle \texttt{Spain, 34}\rangle\}\,. \tag{2.1}$$

Informally then, the *information extraction task* here is to extract this set of tuples from the example query response.

We use the phrase *semi-structured* to describe the country/code resource. While the information to be extracted is a set of fragments that have a regular structure, the page also contains irrelevant text. Furthermore, this regularity (*i.e.*, displaying countries in bold face and codes in italic) is not available as a machine-readable specification, but rather is a fortuitous but idiosyncratic aspect of the country/code resource. The phrase "semi-structured" is meant merely to guide one's intuitions about the kinds of information resource in which we are interested, and so we will not provide a precise definition.

To extract the tuples listed in Equation 2.1, the query response must be parsed to extract the relevant *content*, while discarding the irrelevant text. That is, the tuples in Equation 2.1 correspond to the text fragments that are outlined:

```
<HTML><TITLE>Some Country Codes</TITLE><BODY>
<B>Some Country Codes</B><P>
<B>Congo</B> <I>242</I><BR>
<B>Egypt</B> <I>20</I><BR>
<B>Belize</B> <I>501</I><BR>
<B>Spain</B> <I>34</I><BR>
<HR><B>End</B></BODY></HTML>
```

Before proceeding, note that this framework is not fully general. The content of some query responses might not correspond simply to a set of fragments of the text. For example, if the query responses are natural language texts (*e.g.*, newswire feed), then, depending on the task, the true content may simply not be describable using a simple relational model. (The natural language processing community typically uses the phrase "information extraction" in this richer sense [Cowie & Lehnert 96].) Alternatively, the extracted strings may need to be post-processed (*e.g.*, to remove extraneous punctuation). While these problems are important, we focus on information resources for which the content can be captured exactly as a set of fragments of the raw query response.

In this thesis we are concerned mainly with *tabular* information resources.

Roughly, a resource is tabular if the attributes and tuples never overlap, and if the attributes occur in a consistent order within each tuple. As shown, the country/code resource is tabular. (While we emphasize tabular layouts, note that in Chapter 5, we discuss information extraction from pages with a hierarchically nested structure, such as a book's table of contents.)

Finally, a *wrapper* is a procedure for extracting information from a particular resource. Formally, a wrapper takes as input a query response, and returns as output the set of tuples describing the response's information content.

For example, Figure 2.2 shows the ExtractCCs procedure, a wrapper for the country/code information resource. The wrapper operates by first skipping over the response's head (indicated by the string `<P>`), and then using the strings `<B>`, `</B>`, `<I>` and `</I>` to delimit the left and right sides of the country and code values. Specifically, the tuples are extracted starting at the top of the page; within each tuple the country is extracted first followed by the code. Extraction stops when the tail of the query response is encountered, indicated by `<HR>`.

In Chapter 4, we describe ExtractCCs in detail. We explain why it works, and argue that it is correct; specifically, we discuss why the head and tail of the page must be handled carefully. For now, the point is simply that the ExtractCCs wrapper can be used to extract the information content of responses to queries posed to the country/code resource.

To summarize, we have informally described our model of information extraction. When *queried*, an information *resource* returns a *response*. The information *content* of a response is comprised of specific literal fragments of the response; the remaining irrelevant text is discarded during extraction. We adopt a relational model of a response's information content: the content is taken to be a set of one or more *tuples*, where each tuple consists of a value for each of a fixed set of attributes. The objects in a relational database can be thought of as rows in a table, and so we require that a page's information content be embedded in the responses in a *tabular* layout.

```
ExtractCCs(page $P$)
    skip past first occurrence of <P> in $P$
    while the next occurrence of <B> is before the next occurrence of <HR> in $P$
        for each $\langle \ell_k, r_k \rangle \in \{\langle$<B>, </B>$\rangle, \langle$<I>, </I>$\rangle\}$
            extract from $P$ the value of the next instance of the $k^{\underline{\text{th}}}$ attribute
            between the next occurrence of $\ell_k$ and the subsequent occurrence of $r_k$
    return all extracted tuples
```

Figure 2.2: *The* ExtractCCs *procedure, a wrapper for the country/code resource shown in Figure 2.1.*

Finally, a *wrapper* is a procedure for extracting the relational content from a page while discarding the irrelevant text.

2.3 The formalism

Resources, queries, and responses. An *information resource* $\mathcal{S}$ is a function from a *query* Q to a *response* P:

- The query Q describes the desired information, by means of an expression in some query language $\mathcal{Q}$. For example, $\mathcal{Q}$ might be SQL [ANSI 92] or KQML [Finin et al. 94]. For typical Internet resources, the query is represented by the arguments to a CGI script [hoohoo.ncsa.uiuc.edu/cgi].
- The query response P is the resource's answer to the query. We assume that the response is encoded as a string over some alphabet Σ. Typically, Σ is the ASCII character set, and the responses are HTML pages or unstructured natural language text; alternatively, the responses might obey a standard such as KIF [logic.stanford.edu/kif] or XML [www.w3.org/TR/WD-xml].

Information resource $\mathcal{S}$ can thus be described formally as a function of the form $\mathcal{S} : \mathcal{Q} \rightarrow \Sigma^*$:

$$\underset{Q \in \mathcal{Q}}{\text{query}} \Longrightarrow \boxed{\text{information resource } \mathcal{S}} \Longrightarrow \underset{P \in \Sigma^*}{\text{response}}$$

Given our focus on information extraction, in this thesis we are concerned primarily with responses, rather than the queries. The intent is that our information extraction techniques can be largely decoupled from issues related to queries. For this reason, in the remainder of this thesis, we ignore the query language $\mathcal{Q}$. Under this simplification, resource $\mathcal{S}$ is equivalent to the set of responses it gives to all queries.

Attributes and tuples. We adopt a standard relational data model. Associated with every information resource is a set of K distinct *attributes*, each representing a column in the relational model. In the country/code example, there are $K = 2$ attributes.

A *tuple* is a vector $\langle A_1, \ldots, A_K \rangle$ of K strings; $A_k \in \Sigma^*$ for each k. The string A_k is the *value* of $k^{\underline{\text{th}}}$ attribute. Whereas attributes represent columns in the relational model, tuples represent rows.

Content and labels. The *content* of a page is the set of tuples it contains. For example, Equation 2.1 lists the content of the country/code example page. This notation is adequate, but since pages have unbounded length, we use instead a cleaner and more concise representation of a page's content. The idea is that, rather than listing the attribute value fragments explicitly, a page's *label* represents the content in terms of a set of indices into the page.

For example, the label for the country/code page in Figure 2.1(c) is:

$$\left\{ \begin{array}{c} \Big\langle \langle 78, 83 \rangle,\ \langle 91, 94 \rangle \Big\rangle, \\ \Big\langle \langle 106, 111 \rangle,\ \langle 119, 121 \rangle \Big\rangle, \\ \Big\langle \langle 133, 139 \rangle,\ \langle 147, 150 \rangle \Big\rangle, \\ \Big\langle \langle 162, 167 \rangle,\ \langle 175, 177 \rangle \Big\rangle \end{array} \right\}. \tag{2.2}$$

To understand this label, compare it to Equation 2.1. Equation 2.2 contains four tuples, each tuple consists of $K = 2$ attributes values, and each such value is represented by a pair of integers. Consider the first pair, $\langle 78, 83 \rangle$. These integers indicate that the first attribute of the first tuple is the substring between positions 78 and 83 (*i.e.*, the string `Congo`); inspection of Figure 2.1(c) reveals that these integers are correct. Similarly, the last pair, $\langle 175, 177\rangle$, indicates that the last attribute's country code occurs between positions 175 and 177 (*i.e.*, the string `34`).

More generally, the content of page P is represented as the label

$$L = \left\{ \begin{array}{ccccc} \Big\langle\langle b_{1,1}, e_{1,1}\rangle, & \ldots, & \langle b_{1,k}, e_{1,k}\rangle, & \ldots, & \langle b_{1,K}, e_{1,K}\rangle\Big\rangle, \\ & & \vdots & & \\ \Big\langle\langle b_{m,1}, e_{m,1}\rangle, & \ldots, & \langle b_{m,k}, e_{m,k}\rangle, & \ldots, & \langle b_{m,K}, e_{m,K}\rangle\Big\rangle, \\ & & \vdots & & \\ \Big\langle\langle b_{M,1}, e_{M,1}\rangle, & \ldots, & \langle b_{M,k}, e_{M,k}\rangle, & \ldots, & \langle b_{M,K}, e_{M,K}\rangle\Big\rangle \end{array} \right\}. \qquad (2.3)$$

Label L encodes the content of page P. The page contains $M > 0$ tuples, each of which has $K > 0$ attributes. The integers $1 \leq k \leq K$ are the attributes indices, while the integers $1 \leq m \leq M$ index tuples within the page. Each pair $\langle b_{m,k}, e_{m,k}\rangle$ encodes a single attribute value. The value $b_{m,k}$ is the index in P of the *b̲eginning* of the $k^{\underline{\text{th}}}$ attribute value in the $m^{\underline{\text{th}}}$ tuple. Similarly, $e_{m,k}$ is *e̲nd* index of the $k^{\underline{\text{th}}}$ attribute value in the $m^{\underline{\text{th}}}$ tuple. Thus, the $k^{\underline{\text{th}}}$ attribute of the $m^{\underline{\text{th}}}$ tuple occurs between positions $b_{m,k}$ and $e_{m,k}$ of page P. For example, the pair $\langle b_{3,2}, e_{3,2}\rangle = \langle 147, 150\rangle$ in Equation 2.2 encodes the third tuple's second (country code) attribute in the example page.

Note that missing values are not permitted: each tuple must assign exactly one value to each attribute.

Tabular layout. We are concerned with *tabular* information resources. This requirement amounts to a set of constraints over the $b_{m,k}$ and $e_{m,k}$ indices of legal labels. To be valid, the indices must extract from the page a set of strings that correspond to a tabular layout of the content within the page. For example, if $e_{m,k} > b_{m,k+1}$ for some m and k, then the content of the query response simply can not be arranged in a tabular manner, because the two values occur out of order. The full details[1] are conceptually straightforward but uninteresting.

[1] For a label to be tabular, the following four conditions must hold. (1) The individual fragments are legitimate: $\forall_{m,k}\ b_{m,k} \leq e_{m,k}$. (2) The content consists of disjoint fragments:

The symbol $\mathcal{L}$ refers to the set of all labels. Again, the details[2] are tedious.

Wrappers. Finally, we are in a position to give a precise definition of a wrapper. Formally, a wrapper (*e.g.*, the ExtractCCs procedure) is a function from a query response to a label:

$$\begin{array}{c}\text{query response}\\ P \in \Sigma^*\end{array} \Longrightarrow \boxed{\text{wrapper}} \Longrightarrow \begin{array}{c}\text{label}\\ L \in \mathcal{L}\end{array}$$

At this level of abstraction, a wrapper is simply an arbitrary procedure. Of course, in the remainder of this thesis, we devote considerable attention to particular classes of wrappers.

2.4 Summary

We have presented a simple model of information extraction. *Information resources* return *pages* whose tabular *contents* can be captured in terms of a *label*, a structure using a page's indices to represent the text fragments to be extracted. A *wrapper* is simply a procedure that computes a label for a given page.

With this background in place, we can begin our investigation of wrapper induction. We start in Chapter 3 with a discussion of the inductive learning framework, and show how it can be applied to the problem of automatically constructing wrappers.

$\forall_{m,k}\ \neg\exists_{m',k'}\ (b_{m,k} \leq b_{m',k'} \leq e_{m,k}) \vee (b_{m,k} \leq e_{m',k'} \leq e_{m,k})$. (3) All attributes for one tuple precede all attributes for the next: $\forall_{m<M}\ e_{m,K} < b_{m+1,1}$. (4) Attributes occur sequentially within each tuple: $\forall_{m,k<K}\ e_{m,k} < b_{m,k+1}$.

[2] $\mathcal{L} = \cup_{m>0}\mathcal{L}_m$, where $\mathcal{L}_m$ is the set of labels containing exactly m tuples. Informally, $\mathcal{L}_m = \{L|$ label L contains m tuples and respects the tabularity conditions in Footnote 1$\}$.

Chapter 3

WRAPPER CONSTRUCTION AS INDUCTIVE LEARNING

3.1 Introduction

Our goal is to automatically construct wrappers. In the previous chapter, we described a model of information extraction, which included the specification of a wrapper. Our technique for generating such wrappers is based on inductive learning. We first describe the inductive learning framework (Section 3.2). We then show how to treat wrapper construction as an induction task (Section 3.3).

As we will see, the input to our wrapper construction system is essentially a sample of the behavior of the wrapper to be learned. Under this formulation, wrapper construction becomes a process of reconstructing a wrapper based on a sample of its behavior.

3.2 Inductive learning

Inductive learning has received considerable attention in the machine learning community; see [Angluin & Smith 83] or [Mitchell 97, Chapters 2–3] for surveys. At the highest level, inductive learning is the task of computing, from a set of examples of some unknown target concept, a generalization that (in some domain-specific sense) explains the observations. The idea is that a generalization is good if it explains the observed examples and (more importantly) makes accurate predictions when additional previously-unseen examples are encountered.

For example, suppose an inductive learning system is told that '*Thatcher lied*', '*Mao lied*', and '*Einstein didn't lie*'. The learner might then hypothesize that the general rule underlying the observations is '*Politicians lie*'. This assertion is reasonable, because it is consistent with the examples seen so far. If asked '*Did Nixon lie?*', the learner would then presumably respond '*Yes*'.

We proceed by presenting a formal framework for discussing inductive learning (Section 3.2.1). We then describe Induce, a generic inductive learning algorithm that can be customized to learn a particular hypothesis class (Section 3.2.2). We go on to describe the PAC model, a statistical technique for predicting when the learner has seen enough examples to generate a reliable hypothesis (Section 3.2.3). Finally, we describe the few minor ways in which our treatment of induction is somewhat unconventional (Section 3.2.4).

3.2.1 The formalism

An inductive learning task consists of three elements:

- a set $\mathcal{I} = \{\ldots, I, \ldots\}$ of *instances*;
- a set $\mathcal{L} = \{\ldots, L, \ldots\}$ of *labels*; and
- a set $\mathcal{H} = \{\ldots, H, \ldots\}$ of *hypotheses*.

Each hypothesis $H \in \mathcal{H}$ is a function from $\mathcal{I}$ to $\mathcal{L}$, with the notation

$$H(I) = L$$

indicating that hypothesis $H \in \mathcal{H}$ assigns label $L \in \mathcal{L}$ to instance $I \in \mathcal{I}$.

In the politician example, the set of instances $\mathcal{I}$ captures the set of people about whom assertions are made, and the label set $\mathcal{L}$ indicates whether a particular person is a liar:

$$\begin{aligned} \mathcal{I} &= \{\text{Thatcher, Mao, Einstein, Nixon, } \ldots\} \\ \mathcal{L} &= \{\text{liar, truthful}\} \end{aligned}$$

Since hypotheses are functions from $\mathcal{I}$ to $\mathcal{L}$, the hypothesis '*Men lie*' corresponds to the function that classifies any male as 'liar'.

The idea is that the learner wants to identify some unknown *target* hypothesis $\mathcal{T} \in \mathcal{H}$. Of course, the learner does not have access to $\mathcal{T}$. Instead, the learner observes a set $\mathcal{E}$ of *examples*: $\mathcal{E} = \{\ldots, \langle I, \mathcal{T}(I)\rangle, \ldots\} \subset 2^{\mathcal{I}\times\mathcal{L}}$ is a sample of the instances, each paired with its label according to $\mathcal{T}$. The learner's job is to reconstruct the target $\mathcal{T}$ from the sample $\mathcal{E}$.

This notion of a supply of examples of $\mathcal{T}$ is formalized in terms of an *oracle* that supplies labeled instances. In this simple model of induction, we assume that the learner has no control over which examples are returned by the oracle. Formally, an inductive learning algorithm is given access to a function $\mathsf{Oracle}_{\mathcal{T}}$. $\mathsf{Oracle}_{\mathcal{T}}$ is a function taking no arguments and returning a pair $\langle I, \mathcal{T}(I)\rangle$, where $I \in \mathcal{I}$ and $\mathcal{T}(I) \in \mathcal{L}$. The subscript $\mathcal{T}$ on the oracle function emphasizes that the oracle depends on the target $\mathcal{T}$. The inductive learner calls $\mathsf{Oracle}_{\mathcal{T}}$ repeatedly in order to accumulate the set $\mathcal{E}$ of examples.

3.2.2 The Induce *algorithm*

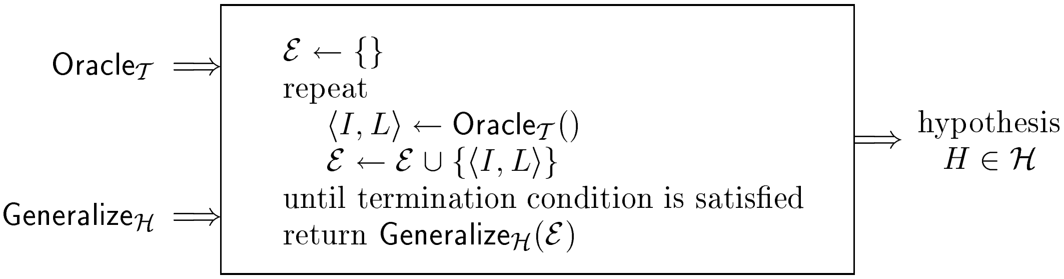
An inductive learning algorithm can be characterized as follows: the algorithm takes as input the oracle function $\mathsf{Oracle}_{\mathcal{T}}$, and outputs an hypothesis $H \in \mathcal{H}$. However, in the remainder of this thesis it will be helpful to open up this "black box" specification, in order to concentrate on learning algorithms that operate in a specific manner.

In particular, Figure 3.1 presents the Induce generic inductive learning algorithm. Induce takes two inputs:

- the oracle function $\mathsf{Oracle}_{\mathcal{T}}$; and
- a generalization function $\mathsf{Generalize}_{\mathcal{H}}$, specific to the hypothesis class $\mathcal{H}$ being learned.

Given these two inputs, Induce must output an hypothesis $H \in \mathcal{H}$.

Note that Induce is a "generic" inductive learning algorithm in the sense that it can be customized to learn a target $\mathcal{T}$ in any hypothesis class $\mathcal{H}$ by supplying the algorithm with the appropriate oracle and generalization functions as input. The oracle


$\mathsf{Oracle}_{\mathcal{T}}$
$\mathsf{Generalize}_{\mathcal{H}}$
hypothesis
$H \in \mathcal{H}$

```
$\mathcal{E} \leftarrow \{\}$
repeat
    $\langle I, L\rangle \leftarrow \mathsf{Oracle}_{\mathcal{T}}()$
    $\mathcal{E} \leftarrow \mathcal{E} \cup \{\langle I, L\rangle\}$
until termination condition is satisfied
return $\mathsf{Generalize}_{\mathcal{H}}(\mathcal{E})$
```

Figure 3.1: *The* Induce *generic inductive learning algorithm (preliminary version; see Figure 3.3).*

function $\mathsf{Oracle}_{\mathcal{T}}$ was just discussed; in the remainder of this section, we describe how Induce works, which will lead to a description of the second input, the $\mathsf{Generalize}_{\mathcal{H}}$ function.

The Induce algorithm operates as follows. First, Induce uses the $\mathsf{Oracle}_{\mathcal{T}}$ function to accumulate a set $\mathcal{E} = \{\ldots, \langle I, L\rangle, \ldots\}$ of examples. Unspecified in Figure 3.1 is the termination condition describing when enough examples have been gathered. As will be described in Section 3.2.3, this termination condition is important for ensuring that Induce's output is satisfactory.

After gathering the examples $\mathcal{E}$, Induce then passes $\mathcal{E}$ to the $\mathsf{Generalize}_{\mathcal{H}}$ function. Recall that $\mathsf{Generalize}_{\mathcal{H}}$ is a domain-specific function, specialized to the hypothesis class $\mathcal{H}$ under consideration. Let us emphasize again that Induce is a generic learner, with the domain-specific "heavy lifting" being done by $\mathsf{Generalize}_{\mathcal{H}}$. Formally, $\mathsf{Generalize}_{\mathcal{H}}$ takes as input a set of examples $\mathcal{E}$, and returns an hypothesis $H \in \mathcal{H}$; thus the generalization function has the form $\mathsf{Generalize}_{\mathcal{H}} : 2^{\mathcal{I}\times\mathcal{L}} \rightarrow \mathcal{H}$.

We now illustrate Induce using a generalization of the politician example introduced earlier. Consider learning the hypothesis class corresponding to all conjunctions of Boolean literals[1] (CBL) composed from N binary features $x_1, x_2, \ldots, x_N$. The fea-

[1] The *literals* of proposition x are x and its complement $\bar{x}$.

tures define the set of instances: $\mathcal{I} = \{\text{TRUE}, \text{FALSE}\}^N$. Instances are classified in just two ways: $\mathcal{L} = \{\text{TRUE}, \text{FALSE}\}$. The hypothesis class $\mathcal{H}_{\text{CBL}}$ consists of conjunctions of the $2N$ literals: $\mathcal{H}_{\text{CBL}} = \{v_1 \wedge v_2 \wedge \cdots \mid \text{each } v_n \text{ is a literal}\}$. In the politicians example, x_1 might indicate that a person is a male; x_2, that they have red hair; and x_3, that they are a politician. Each person could then represented by a Boolean assignment to each variable, and the label TRUE might indicate that the person doesn't lie while FALSE indicates the person is a liar.

To use the Induce algorithm to learn $\mathcal{H}_{\text{CBL}}$, we must provide it with two inputs, $\text{Oracle}_{\mathcal{T}}$ and $\text{Generalize}_{\text{CBL}}$. For now, we assume that the oracle function is built using whatever means are appropriate to the particular learning task; for example, a person might act as the oracle.

The second input is the generalization function $\text{Generalize}_{\text{CBL}}$:

```
Generalize_CBL(examples $\mathcal{E}$)
    $H \leftarrow x_1 \wedge \bar{x}_1 \wedge \cdots \wedge x_N \wedge \bar{x}_N$
    for each example $\langle I, \text{TRUE} \rangle \in \mathcal{E}$
        for each feature $x_n$
            if $x_n = \text{TRUE}$ in instance $I$, then remove $\bar{x}_n$ from $H$
            else remove $x_n$ from $H$
    return $H$
```

$\text{Generalize}_{\text{CBL}}$ starts with an hypothesis containing every literal, and then eliminates the literals that are inconsistent with the TRUE instances (FALSE instances are simply ignored).

Is $\text{Generalize}_{\text{CBL}}$ a "good" generalization function? As we'll see next, the answer to this question is tightly coupled to Induce's termination condition, left unspecified in Figure 3.1.

3.2.3 PAC analysis

The $\text{Generalize}_{\text{CBL}}$ function demonstrates how the Induce generic induction algorithm is customized to a particular hypothesis class, $\mathcal{H}_{\text{CBL}}$ in this case. But so far we have neglected a crucial question. What reason is there to believe that the stated

Generalize$_{\mathrm{CBL}}$ function ouputs good hypotheses? Why should we believe it will perform better than, for example, simply returning the hypothesis that classifies every instance as TRUE?

Note that the answer is directly relevant to Induce's termination condition, left unspecified in Figure 3.1. This termination condition governs how many examples Induce accumulates into the set $\mathcal{E}$ before invoking the generalization function Generalize$_{\mathcal{H}}$. In this section, I describe one particular termination criterion, based on the *probably approximately correct* (PAC) model [Valiant 84, Kearns & Vazirani 94, Martin & Biggs 92].

Ideally, we want the Induce learning algorithm to output the target hypothesis—*i.e.*, the function actually used to label the examples. However, since the target hypothesis is unknown, this intuition does not yield a satisfactory evaluation criterion.

A more sophisticated alternative involves estimating the performance of an hypothesis on instances other than the examples. By definition, the target will perform perfectly on future instances; we want Induce to output an hypothesis that is expected to perform well on future instances too. PAC analysis is a technique for estimating the expected future performance of an hypothesis; this estimate is based on the examples from which the hypothesis was generated.

The PAC model adopts a statistical approach to evaluating a learner. PAC analysis starts with the observation that, in many learning tasks, it makes sense to assume that both the testing and training instances are distributed according to a fixed but unknown and arbitrary probability distribution. Thus, the extent to which an hypothesis performs well with respect to this distribution can be estimated from its performance on training data drawn from the same distribution. Specifically, the PAC model uses as a performance measure the probability that the learned hypothesis is correct.

The PAC model is formalized as follows.

- Let the instances $\mathcal{I}$ be distributed according to $\mathcal{D}$, a fixed but unknown and arbitrary probability distribution over $\mathcal{I}$. We will be interested in the chance of drawing an instance with particular properties: the probability of selecting from an instance I with property q is written $\mathcal{D}[I|q(I)]$.

- The example oracle $\mathsf{Oracle}_{\mathcal{T}}$ is modified so that it returns instances distributed according to $\mathcal{D}$. We use the notation $\mathsf{Oracle}_{\mathcal{T},\mathcal{D}}$ to emphasize this dependancy.
- The *error* $E_{\mathcal{T},\mathcal{D}}(H)$ of hypothesis $H \in \mathcal{H}$ is the probability (with respect to $\mathcal{D}$) that H and $\mathcal{T}$ disagree about a single instance drawn randomly from D:

$$E_{\mathcal{T},\mathcal{D}}(H) = \mathcal{D}\left[I|H(I) \neq \mathcal{T}(I)\right]$$

- Finally, let $0 < \epsilon < 1$ be a parameter describe the desired *accuracy*, and $0 < \delta < 1$ be a second parameter, the desired *reliability*. The meaning of these two parameters will be described shortly.

The idea of the PAC model is that the learner Induce invokes $\mathsf{Oracle}_{\mathcal{T},\mathcal{D}}$ as many times as needed to be *reasonably sure* that its hypothesis will have *low error*. In general, we expect that the error of the hypotheses generated by Induce will decreases with the number of training examples. This intuition is formalized statistically as follows:

> **PAC termination criterion:** An inductive learning algorithm for hypothesis class $\mathcal{H}$ should examine as many examples as are needed in order to ensure that, for any target $\mathcal{T} \in \mathcal{H}$, distribution $\mathcal{D}$, and $0 < \epsilon,\ \delta < 1$, the algorithm will output an hypothesis $H \in \mathcal{H}$ satisfying $E_{\mathcal{T},\mathcal{D}}(H) < \epsilon$, with probability at least $1 - \delta$.

Under this formalization, the parameters ϵ and δ define what "low error" and "reasonably sure" (respectively) mean: the learned hypothesis must have error bounded by ϵ, and this must happen with probability at least $1 - \delta$. Note that the learner must perform increasingly well as ϵ and δ approach zero.

As illustrated in Figure 3.2, the two parameters ϵ and δ are needed to handle the two kinds of difficulties Induce may encounter. The reliability parameter δ is required because the PAC termination criterion does not involve the *particular* examples in $\mathcal{E}$,

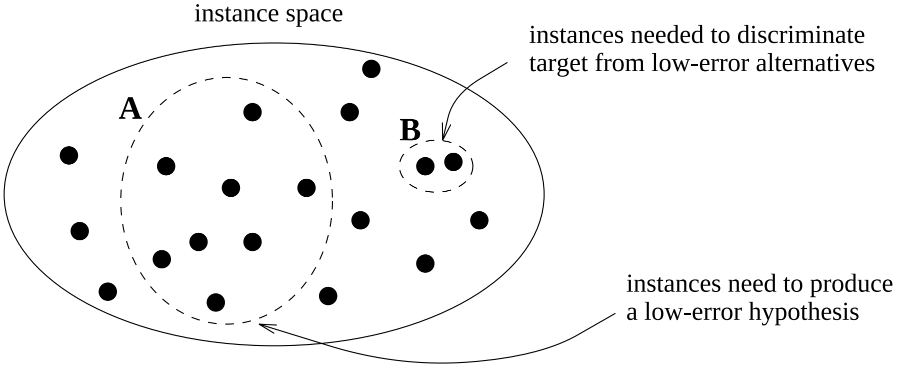

instance space
instances needed to discriminate target from low-error alternatives
A
B
instances need to produce a low-error hypothesis

Figure 3.2: *Two parameters, ϵ and δ, are needed to handle the two types of difficulties that may occur while gathering the examples $\mathcal{E}$.*

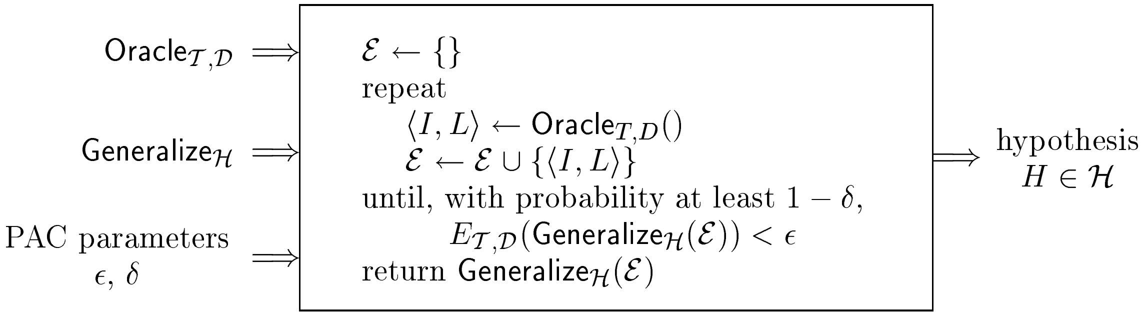
and thus Induce could unluckily receive from $\mathsf{Oracle}_{\mathcal{T},\mathcal{D}}$ an unrepresentative sample of the instances. In Figure 3.2, the learner may be unlucky and see no instances from the region marked '**A**'. In summary, there is always some chance that the error of the learned hypothesis will be large.

On the other hand, the accuracy parameter ϵ is required because $\mathcal{D}$ might be structured so that there is only a small chance under $\mathcal{D}$ of encountering an example needed to discriminate between the target $\mathcal{T}$ and alternative low-error hypotheses. In Figure 3.2, there is only a small chance of seeing the instances in the region marked '**B**'. To summarize, it is unlikely that the error will be exactly zero.

Figure 3.3 illustrates how to incorporate the PAC model into the Induce algorithm. This revised version of Induce takes as input parameters ϵ and δ, and the algorithm stops gatherings examples when they satisfy the PAC termination criterion.

In general, whether the PAC termination criterion can always be satisfied depends on the hypothesis class $\mathcal{H}$. If such a guarantee can be made, then (following [Kearns & Vazirani 94, Definition 1]) we say that hypothesis class $\mathcal{H}$ is *PAC-learnable.*

> **Definition 3.1 (PAC-learnable)** *Hypothesis class $\mathcal{H}$ is* PAC-learnable *iff there exists a generalization function $\mathsf{Generalize}_{\mathcal{H}}$ with the following property: for every target hypothesis $\mathcal{T} \in \mathcal{H}$, for every distribution $\mathcal{D}$ over*


$\mathsf{Oracle}_{\mathcal{T},\mathcal{D}}$
$\mathsf{Generalize}_{\mathcal{H}}$
PAC parameters
ϵ, δ
hypothesis
$H \in \mathcal{H}$

```
$\mathcal{E} \leftarrow \{\}$
repeat
    $\langle I, L\rangle \leftarrow \mathsf{Oracle}_{\mathcal{T},\mathcal{D}}()$
    $\mathcal{E} \leftarrow \mathcal{E} \cup \{\langle I, L\rangle\}$
until, with probability at least $1-\delta$,
        $E_{\mathcal{T},\mathcal{D}}(\mathsf{Generalize}_{\mathcal{H}}(\mathcal{E})) < \epsilon$
return $\mathsf{Generalize}_{\mathcal{H}}(\mathcal{E})$
```

Figure 3.3: *A revised version of* Induce*; see Figure 3.1.*

the instances $\mathcal{I}$, and for all $0 < \epsilon < 1$ and $0 < \delta < 1$, if Induce *is given as input* $\mathsf{Generalize}_{\mathcal{H}}$*,* $\mathsf{Oracle}_{\mathcal{T},\mathcal{D}}$*, ϵ, and δ, then, with probability at least $1-\delta$,* Induce *will output an hypothesis $H \in \mathcal{H}$ satisfying $E_{\mathcal{T},\mathcal{D}}(H) < \epsilon$.*

If in addition $\mathsf{Generalize}_{\mathcal{H}}$ *runs in time polynomial in $\frac{1}{\epsilon}$, $\frac{1}{\delta}$, and the size of the instances*[2]*, then we say that $\mathcal{H}$ is* efficiently *PAC-learnable.*

The standard technique for establishing that class $\mathcal{H}$ is PAC-learnable is to determine, given values of ϵ and δ (but with no knowledge of $\mathcal{T}$ or $\mathcal{D}$), the least value B such that if $|\mathcal{E}| \geq B$, then the PAC termination criterion will hold. B is a lower bound on the number of examples needed to satisfy the PAC termination criterion. In general, B depends on ϵ and δ; we write $B = B(\epsilon, \delta)$ to emphasize this relationship. The Induce learning algorithm uses this bound $B(\epsilon, \delta)$ to decide when it has gathered enough examples; $B(\epsilon, \delta)$ serves that the termination conditions, unspecified

[2] We are sweeping a subtlety of the PAC model under the rug here. Recall the class $\mathcal{H}_{\mathrm{CBL}}$ over instances defined by N binary features. We ought to give a learning algorithm more time as N increases. For example, N is clearly a lower bound on the time to read a single instance, as well as the time to write out the final hypothesis. Thus N captures the "complexity" of learning $\mathcal{H}_{\mathrm{CBL}}$. More generally, for any particular learning task, we want to allow the learner algorithm additional time as the task's natural complexity measure increases. We will return to this issue in Section 4.6, when we apply the PAC model to the task of learning HLRT wrappers. See [Kearns & Vazirani 94, Section 1.2.2] for a thorough discussion.

in Figure 3.1.

For example, recall the politician example and $\mathcal{H}_{\mathrm{CBL}}$ hypothesis class introduced earlier. It is straightforward to show [Kearns & Vazirani 94, Theorem 1.2] that for all ϵ, δ, $\mathcal{D}$, and $\mathcal{T}$, if Induce gathers a set of examples $\mathcal{E}$ such that

$$|\mathcal{E}| \geq \frac{2N}{\epsilon}\left(\ln 2N + \ln\frac{1}{\delta}\right),$$

(where the instances are defined by N binary features) then the PAC termination criterion is satisfied. We conclude that $\mathcal{H}_{\mathrm{CBL}}$ is PAC-learnable, because Induce can use the value $B(\epsilon,\delta) = \frac{2N}{\epsilon}(\ln 2N + \ln\frac{1}{\delta})$ as its termination condition. Furthermore, note that (1) $B(\epsilon,\delta)$ is polynomial in N, $\frac{1}{\epsilon}$, and $\frac{1}{\delta}$, and (2) the Generalize$_{\mathrm{CBL}}$ function runs in time polynomial in $|\mathcal{E}|$ and N; therefore we conclude that $\mathcal{H}_{\mathrm{CBL}}$ is efficiently PAC-learnable.

3.2.4 Departures from the standard presentation

The formal framework for inductive learning presented in this chapter differs somewhat from the "standard" presentation usually found in the machine learning literature (*e.g.*, [Mitchell 97]). Formally speaking, these discrepancies are relatively unimportant. Throughout, our objective is to simplify as much as possible our notation and formalism. In some cases this objective has lead to describing inductive learning in more general terms than usual; in other cases we have presented a relatively restricted notion of induction. In this section we briefly highlight the differences between the standard presentation and ours.

First, we present the instances simply as a *set* $\mathcal{I}$, rather than the usual practice of describing $\mathcal{I}$ as the *space* induced by a given language for representing instances. Similarly, we define $\mathcal{H}$ as a set of functions, rather than as the space induced by an hypothesis representation language. Of course, in order to actually apply the inductive framework to a particular learning task, one must select representation languages for the instances and hypotheses, and the key to successful learning is invariably to *bias* [Mitchell 80] the learning algorithm so that it considers only hypotheses in a carefully crafted space.

For example, the CBL example was described by specifying that $\mathcal{I} = \{\text{TRUE}, \text{FALSE}\}^N$ (for some fixed N) and $\mathcal{H}_{\text{CBL}} = \{v_1 \wedge v_2 \wedge \cdots \mid \text{each } v_n \text{ is a literal}\}$; we have seen that CBL is efficiently learnable. However, this thesis is concerned with information extraction and wrapper induction, and it turns out that the representation languages that have been examined are not appropriate for our application. In Chapters 4 and 5, we describe wrapper induction by precisely specifying how to bias the Induce inductive learning algorithm in order to make wrapper induction feasible.

Second, the standard presentation assumes that the set $\mathcal{L}$ of labels has fixed cardinality; often, $\mathcal{L}$ is simply assumed to be the set of Boolean values $\{\text{TRUE}, \text{FALSE}\}$. For example, *concept learning* is a well-studied special case of induction [Mitchell 97, Chapter 2]; a concept is equated with the set of instances in its extension or (equivalently) the function which maps instances of the concept to TRUE and other instances to FALSE. In contrast, we permit $\mathcal{L}$ to have arbitrary (including infinite) cardinality. As we'll see, allowing $\mathcal{L}$ to have arbitrary cardinality is essential to formalizing wrapper construction as an induction task.

Third, we assumed that the learner gains knowledge of the target hypothesis $\mathcal{T}$ only by means of the oracle $\mathsf{Oracle}_{\mathcal{T},\mathcal{D}}$. In this thesis we ignore other kinds of queries to the oracle, such as equivalence and membership queries [Angluin 87, Angluin & Laird 88]. Furthermore, we have considered only perfect, noise-free oracles—*i.e.*, invoking $\mathsf{Oracle}_{\mathcal{T},\mathcal{D}}$ returns exactly $\langle I, \mathcal{T}(I)\rangle$ (for some instance I), rather than sometimes reporting $\mathcal{T}(I)$ incorrectly. We will continue to assume noise-free oracles until Chapter 6, when we consider extensions to this simple model in the context of wrapper induction.

Fourth, even assuming simple, noise-free oracles, the Induce algorithm is highly specialized to the task at hand. Induce simply gathers a set $\mathcal{E}$ of examples, and calls the generalization function $\mathsf{Generalize}_{\mathcal{H}}$ once with input $\mathcal{E}$. In the context of wrapper induction or learning CBL, this simple algorithm is sufficient. But other strategies might be useful in other learning tasks. For example, *boosting* algorithms [Kearns & Vazirani 94, Chapter 4] invoke $\mathsf{Generalize}_{\mathcal{H}}$ several times with different sets of gathered examples, and output the hypothesis that performs best against additional examples from $\mathsf{Oracle}_{\mathcal{T},\mathcal{D}}$.

Note also that, unlike most descriptions of induction, we explicitly decompose the learning algorithm into "generic" learner Induce, and a "domain-specific" generalization function $\mathsf{Generalize}_{\mathcal{H}}$. We do this because it will be convenient to be able to simply "plug in" generalization functions for different wrapper classes.

Finally, the definition of PAC learnability (Definition 3.1) is stated in terms of the Induce algorithm's $\mathsf{Generalize}_{\mathcal{H}}$ input. Usually, PAC learnability is defined by saying that there must exist a learning algorithm with a particular property (namely, that the PAC termination criterion is satisfied for any ϵ, δ, $\mathcal{T}$, and $\mathcal{D}$). Definition 3.1 is simply a restricted version of this more general definition, specialized to the case when the learning algorithm is Induce. Clearly, if $\mathcal{H}$ is PAC-learnable according to our Definition 3.1, then it is PAC-learnable in the more general sense.

3.3 Wrapper construction as inductive learning

This chapter has been concerned with inductive learning in general. With this background in place, we now show how wrapper induction can be viewed as a problem of induction.

Recall that an induction task is comprised of **(a)** a set of instances $\mathcal{I}$; **(b)** a set of labels $\mathcal{L}$; and **(c)** a set of hypotheses $\mathcal{H}$. In our framework, to learn $\mathcal{H}$ we must provide the Induce generic learning algorithm with two inputs: **(d)** the oracle function $\mathsf{Oracle}_{\mathcal{T},\mathcal{D}}$, and **(e)** the function $\mathsf{Generalize}_{\mathcal{H}}$. Finally, need to develop **(f)** a PAC model of learning $\mathcal{H}$.

The correspondence between inductive learning and wrapper induction is as follows:

(a) The instances $\mathcal{I}$ correspond to query responses from the information resource under consideration. In the country/code example described in Chapter 2, $\mathcal{I}$ would be a set of HTML strings similar to Figure 2.1(c).

(b) The labels $\mathcal{L}$ correspond to query response labels. For example, the label of the HTML document in Figure 2.1(c) is shown in Equation 2.2.

Note that in typical inductive learning tasks, labels are simply category assignments such as 'liar' or 'truthful', while our labels are complex multi-dimensional structures. However, a *single* such structure corresponds to a page's label, just as the single label 'liar' is assigned to Nixon. Let us explicitly state that even though our labels are structured, we do *not* assign multiple labels to any instance.

(c) Hypotheses correspond to wrappers, and an hypothesis bias corresponds to a class of wrappers. For example, the ExtractCCs wrapper shown in Figure 2.2 is one candidate hypothesis. Note that ExtractCCs satisfies the formal definition of an hypothesis, because it takes as input a query response and outputs a label. As hinted at in Chapter 1 (and described in detail in Chapter 4) ExtractCCs is a member of the HLRT wrapper class. In Chapter 5, we identify several additional classes of wrappers which are learnable under this framework.

(d) The $\mathsf{Oracle}_{\mathcal{T},\mathcal{D}}$ function produces a labeled example query response for a particular information resource.

The idea is that associated with each resource is a target wrapper $\mathcal{T}$. Of course, in our wrapper construction application, $\mathcal{T}$ usually does not exist until our system learns it. Nevertheless, we can treat the oracle as being dependent on

$\mathcal{T}$. The function $\mathsf{Oracle}_{\mathcal{T},\mathcal{D}}$ need only return an example of how the target *would* behave; the oracle does not require access to the target itself.

As a simple example, a person might play the role of the oracle. Alternatively, we describe in Chapter 6 techniques for automatically labeling pages. These techniques can be used to implement $\mathsf{Oracle}_{\mathcal{T},\mathcal{D}}$ without requiring access to $\mathcal{T}$.

(e) In Chapter 4, we discuss how to implement the $\mathsf{Generalize}_{\mathrm{HLRT}}$, the generalization function for the HLRT wrapper class. Chapter 5 then goes on to define this function for several more classes.

(f) In Chapter 4, we develop a PAC model of our wrapper learning task.

3.4 Summary

Inductive learning is a well-studied model for analyzing and building systems that improve over time or generalize from their experience. The framework provides a rich variety of analytical techniques and algorithmic ideas.

In this chapter, we showed how our wrapper construction task can be viewed as one of induction. To summarize, *query responses* correspond to *instances*, a page's *information content* is its *label*, and *wrappers* correspond to *hypotheses*.

Of course, so far we have only sketched out the mapping between wrapper construction and induction. In Chapter 4 we flesh out the details for one particular wrapper class, HLRT, and in Chapter 5 we consider five additional wrapper classes.

Chapter 4

THE HLRT WRAPPER CLASS

4.1 Introduction

In Chapter 2, we discussed information extraction in general, and using wrappers for information extraction in particular. We then discussed inductive learning, our approach to automatically constructing wrappers. We noted that the key to a successful induction algorithm is invariably to bias the algorithm so that it considers a restricted class of hypotheses [Mitchell 80]. In this chapter, we describe such a restricted class for our application: the HLRT wrapper class is a generalization of the ExtractCCs procedure described in Chapter 2.

We proceed as follows. First, we describe the HLRT class (Section 4.2). We then show how to learn HLRT wrappers, by describing the HLRT-specific generalization function required by the Induce generic learning algorithm. In Section 4.3, we present a straightforward (though inefficient) implementation of this function. We then analyze the computational complexity of our algorithm, and use this analysis to build a more efficient algorithm (Section 4.4). We then analyze our algorithm heuristically in order to understand theoretically why our algorithm runs so quickly in practice (Section 4.5). Finally, in Section 4.6 we develop a PAC model for the HLRT wrapper class.

4.2 HLRT *wrappers*

Recall the wrapper ExtractCCs (Figure 2.2) for the example country/code resource (Figure 2.1). The ExtractCCs wrapper operates by searching for the strings `<B>` and

`</B>` in order to locate and extract the countries, and for the strings `<I>` and `</I>` to extract the country codes.

ExtractCCs is somewhat more complicated, because this simple left-right (LR) strategy fails: the top and the bottom of the page are formatted in such a way that not all occurrences of `<B>`···`</B>` indicate a country. However, the string `<P>` can be used to distinguish the head of the page from its body. Similarly, `<HR>` separates the last tuple from the tail. This more sophisticated head-left-right-tail (HLRT) approach can be used to extract the content from the country/code resource.

The ExecHLRT procedure. In this thesis we focus on HLRT wrappers—informally, HLRT wrappers are those that are structurally similar to ExtractCCs. The idea is that ExtractCCs is one instance of a particular "idiomatic programming style" that is useful when writing wrappers. Figure 4.1 lists ExecHLRT, a template for writing wrappers according to this idiomatic style. ExecHLRT generalizes the ExtractCCs wrapper, by substituting variables in place of the constant strings that are specific to the country/code example, and by allowing K attributes per tuple instead of exactly two. Note that the HLRT delimiters must be constant strings, rather than (for example) regular expressions.

Specifically, the variable h in ExecHLRT represents the head delimeter; $h =$ `<P>` for ExtractCCs. The variable t represents the tail delimiter; $t =$ `<HR>` in the example. There are $K = 2$ attributes in the example. The variable ℓ_1 indicates the string marking the left-hand side of the first attribute (the country); $\ell_1 =$ `<B>` in the example. r_1 marks the right-hand side of the first attribute; $r_1 =$ `</B>` in the example. Finally, ℓ_2 and r_2 mark the left- and right-hand (respectively) sides of the second attribute (the code); $\ell_2 =$ `<I>` and $r_2 =$ `</I>` in the example.

ExecHLRT operates by skipping over the page's head, marked by the value h. Next, the tuples are extracted one by one, stopping when the page's tail (indicated by t) is encountered. The algorithm's outer loop terminates when the tail is encountered.

(a)

```
ExecHLRT(wrapper $\langle h, t, \ell_1, r_1, \ldots, \ell_K, r_K \rangle$, page $P$)
    skip past the first occurrence of $h$ in $P$
    while the next occurrence of $\ell_1$ is before the next occurrence of $t$ in $P$
        for each $\langle \ell_k, r_k \rangle \in \{\langle \ell_1, r_1 \rangle, \ldots, \langle \ell_K, r_K \rangle\}$
            extract from $P$ the value of the next instance of the $k^{\underline{\text{th}}}$ attribute
                between the next occurrence of $\ell_k$ and subsequent occurrence of $r_k$
    return all extracted tuples
```

(b)

```
ExecHLRT(wrapper $\langle h, t, \ell_1, r_1, \ldots, \ell_K, r_K \rangle$, page $P$)
    $i \leftarrow P\#h$                                                    4.1(a)
    $m \leftarrow 0$
    while $|P[i]/t| > |P[i]/\ell_1|$                                       4.1(b)
        $m \leftarrow m + 1$
        for each $\langle \ell_k, r_k \rangle \in \{\langle \ell_1, r_1 \rangle, \ldots, \langle \ell_K, r_K \rangle\}$
            $i \leftarrow i + P[i]\#\ell_k + |\ell_k|$                     4.1(c)
            $b_{m,k} \leftarrow i$                                         4.1(d)
            $i \leftarrow i + P[i]\#r_k$                                   4.1(e)
            $e_{m,k} \leftarrow i - 1$                                     4.1(f)
    return label $\{\ldots, \langle \ldots, \langle b_{m,k}, e_{m,k} \rangle, \ldots \rangle, \ldots\}$    4.1(g)
```

Figure 4.1: *The* HLRT *wrapper procedure template:* (a), *pseudo-code; and* (b), *details.*

The inner loop extract's each of a tuple's attributes, using the left- (ℓ_k) and right-hand (r_k) delimiter for each attribute in turn.

ExecHLRT is essentially a "template" for writing HLRT wrappers. For instance, if we instantiate the ExecHLRT template with the values $K = 2$, $h =$ `<P>`, $t =$ `<HR>`, $\ell_1 =$ `<B>`, $r_1 =$ `</B>`, $\ell_2 =$ `<I>`, $r_2 =$ `</I>`, the result is the ExtractCCs wrapper.

The ExecHLRT procedure: Details. Figure 4.1 presents ExecHLRT at two levels of detail. In part (a) of the figure, we provide a high-level pseudo-code description of the algorithm. In Appendix B, we provide proofs for the theorems and lemmas stated in this thesis. To do so involves reasoning formally about the behavior of the ExecHLRT algorithm. Such reasoning involves specifying the algorithm in more detail than is supplied in Figure 4.1(a); part (b) provides the additional details.

Before proceeding, we briefly discuss these details.

ExecHLRT operates by incrementing an index i over the input page P. The variable i is increased by searching for the HLRT delimiters in the page. The attribute's beginning indices (the $b_{m,k}$) and ending indices (the $e_{m,k}$) are computed based on the values taken by i as the page is processed.

Figure 4.1(b) makes use of the "/" and "#" string operators. In Appendix C, we describe our string algebra; here we provide a brief summary. If s and s' are strings, then s/s' is the suffix of s starting at the first occurrence of s', with $s/s' = \Diamond$ indicating that s' does not occur in s. For example, `abcdecdf/cd` $=$ `cdecdf`, while `abc/xyz` $= \Diamond$. While "/" is a string *search* operator, "#" is a string *index* operator: $s\#s'$ is the position of s' in s; for example `abcdef#cd` $= 3$.

ExecHLRT proceeds as follows. First (line 4.1(a)), the index i points to the head delimiter h in the input page P. For each iteration of the outer 'while' loop (line 4.1(b)) i points to a position upstream of the start of the next tuple. For each iteration of the inner 'for each' loop, i points first (line 4.1(c)) at the beginning of the $m^{\underline{\text{th}}}$ tuple's $k^{\underline{\text{th}}}$ attribute. The variable i then (line 4.1(e)) points to one character beyond the end of the $m^{\underline{\text{th}}}$ tuple's $k^{\underline{\text{th}}}$ attribute. The values of $b_{m,k}$ and $e_{m,k}$ are set using these two indices (lines 4.1(d) and 4.1(f)). The outer loop terminates (line 4.1(b)) when the next occurrence t occurs before the next occurrence of ℓ_1 (if there is one), indicating that P's tail has been encountered.

Formal considerations. Note that we deliberately ignore failure in ExecHLRT. For example, what happens if the head delimiter h does not occur in page P? From a practical perspective, these are important issues (though it is trivial to add code to detect these situations). However, for now this complication need not concern us. As we'll see, we can ignore failure because our induction system does not actually invoke wrappers. Rather, it reasons about what would happen if they were to be invoked; the *consistency constraints* developed in Section 4.3.1 are a precise characterization of the conditions under which ExecHLRT will fail.

The notation presented in Chapter 2 can be used to formally describe the behavior of ExecHLRT. Let w be an HLRT wrapper, P be a page, and L be a label. In Chapter 2 we used the notation $w(P) = L$ to indicate that the L is the result of applying wrapper w to page P. ExecHLRT is simply a procedural description of this relationship: $w(P) = L$ if and only if L is output as a result of invoking ExecHLRT on w and P. That is, the notation $w(P) = L$ is an abbreviation for $\mathsf{ExecHLRT}(w, P) = L$.

Note that, given the ExecHLRT procedure, an HLRT wrapper's behavior can be entirely captured in terms of $2K + 2$ strings h, t, ℓ_1, r_1, …, ℓ_K and r_K (where each tuple consists of K attributes). For example, as described earlier, the six strings ⟨<P>, <HR>, <B>, </B>, <I>, </I>⟩ exactly specify the ExtractCCs wrapper. For this reason, in the remainder of this thesis we treat a wrapper as simply a vector of strings; the ExecHLRT procedure itself plays a relatively minor role. Implicit in the use of the notation $\langle h, t, \ell_1, r_1, \ldots, \ell_K, r_K\rangle$ is the fact that the ExecHLRT procedure defines such a wrapper's behavior.

> **Definition 4.1 (HLRT)** *An* HLRT wrapper *is a vector $\langle h, t, \ell_1, r_1, \ldots, \ell_K, r_K\rangle$ consisting of $2K + 2$ strings, where the parameter K indicates the number of attributes per tuple. The* HLRT wrapper class $\mathcal{H}_{\text{HLRT}}$ *is the set consisting of all* HLRT *wrappers.*[1]

This chapter concerns the HLRT wrapper class; when not stated explicitly, any use of the generic term "wrapper" refers to HLRT wrappers only.

[1] Strictly speaking, each integer K induces a different hypothesis set $\mathcal{H}_{\text{HLRT}}$, and thus $\mathcal{H}_{\text{HLRT}}$ is in fact a function of K. However, the value of K will always be clear from context and thus for simplicity we do not explicitly note this dependency.

4.3 The Generalize$_{\text{HLRT}}$ *algorithm*

We want to use the Induce generic induction algorithm to learn HLRT wrappers. Recall that Induce takes as input a function Generalize$_{\mathcal{H}}$, which is customized to the particular hypothesis class $\mathcal{H}$ being learned. In this section, we describe Generalize$_{\text{HLRT}}$, the generalization function for the $\mathcal{H}_{\text{HLRT}}$ hypothesis class.

Generalize$_{\text{HLRT}}$ takes as input a set $\mathcal{E} = \{\langle P_1, L_1\rangle, \ldots, \langle P_N, L_N\rangle\}$ of examples. Each example is a pair $\langle P_n, L_n\rangle$, where P_n is a page and L_n is its label according to the target wrapper $\mathcal{T}$: $L_n = \mathcal{T}(P_n)$. When invoked on a set of examples, Generalize$_{\text{HLRT}}$ returns a wrapper $\langle h, t, \ell_1, r_1, \ldots, \ell_K, r_K\rangle \in \mathcal{H}_{\text{HLRT}}$.

At a theoretical level, what formal properties should Generalize$_{\text{HLRT}}$ exhibit? In an inductive learning setting, we generally can not insist that the learner be *perfect*—*i.e.*, that it always output the target hypothesis. The reason is simply that, by definition, induction involves drawing inferences that may be invalid. (Of course, the hope is that the inferences will be appropriate if based on a significant amount of training examples.) Rather than perfect, we want our learner to be *consistent*—*i.e.*, it always outputs an hypothesis that is correct with respect to the training examples. As we'll see, consistency is an important formal property not only because it is intuitively reasonable, but also because our PAC analysis of HLRT requires that Generalize$_{\text{HLRT}}(\mathcal{E})$ be consistent.

We therefore begin our description of Generalize$_{\text{HLRT}}$ by describing the conditions under which an HLRT wrapper is consistent with a set of examples. These conditions, which we call the HLRT *consistency constraints*, are used throughout this thesis to understand the nature of HLRT wrapper induction.

The section is organized as follows.

- In Section 4.3.1, we present the HLRT consistency constraints.
- In Section 4.3.2, we present Generalize$_{\text{HLRT}}$, a special-purpose constraint-satisfaction engine, customized to the HLRT consistency constraints.
- In Section 4.3.3, we illustrate Generalize$_{\text{HLRT}}$ by walking through an example.

- Finally, in Section 4.3.4, we prove that $\mathsf{Generalize}_{\mathrm{HLRT}}$ is consistent.

4.3.1 The HLRT *consistency constraints*

We begin with a definition of consistency. Though more general definitions are possible, we will specialize our definition to the use of the Induce generic learning algorithm.

> **Definition 4.2 (Consistency)** *Let $\mathcal{H}$ be an hypothesis class.* $\mathsf{Generalize}_{\mathcal{H}}$ *is* consistent *iff, for every set of examples $\mathcal{E}$, we have that $H(I) = L$ for every $\langle I, L\rangle \in \mathcal{E}$, where $H =$* $\mathsf{Generalize}_{\mathcal{H}}(\mathcal{E})$ *is the hypothesis returned by* $\mathsf{Generalize}_{\mathcal{H}}$ *when given input $\mathcal{E}$.*

To reason about the consistency of $\mathsf{Generalize}_{\mathrm{HLRT}}$, we must apply this definition to the $\mathcal{H}_{\mathrm{HLRT}}$ wrapper class. What conditions must hold for wrapper w to be consistent with a particular example $\langle P, L\rangle$? The HLRT *consistency constraints* provide the answer to this question. By examining the ExecHLRT procedure in detail, we can define a predicate $\mathcal{C}_{\mathrm{HLRT}}(w, \langle P, L\rangle)$ which ensures that ExecHLRT computes L given w and P. That is, $\mathcal{C}_{\mathrm{HLRT}}(w, \langle P, L\rangle)$ provides the necessary and sufficient conditions under which $w(P) = L$.

A precise description of $\mathcal{C}_{\mathrm{HLRT}}$ requires a significant amount of notation. Before tackling these details, we'll illustrate the basic idea using the country/code example (Figure 2.1).

Example. Why does the ExtractCCs wrapper work? Consider a (relatively simple) aspect of this question: Why is `</B>` a satisfactory value for r_1 in the HLRT encoding of the ExtractCCs wrapper? To answer this question, we must examine the ExecHLRT procedure (Figure 4.1).

The variable r_k takes on tihe value $r_1 =$ `</B>` in the first iteration of the inner 'for each' loop. ExecHLRT scans the input page, extracting each attribute value in turn. In particular, for each tuple, ExecHLRT looks for r_1 immediately after the location where it found ℓ_1; the first attribute is extracted between these two indices. Thus r_1 must satisfy two properties: r_1 must occur immediately after each instance of the first attribute in each tuple, and r_1 must *not* occur in any of the instances of the first attribute itself (otherwise, r_1 would occur "upstream", before the true end of the attribute). We can see that `</B>` satisfies both properties: `</B>` is a prefix of the strings occurring after the countries (in this case, each is the string `</B> <I>`), and `</B>` is also not a substring of any of the attribute values (`Congo`, `Egypt`, `Belize`, `Spain`).

On the other hand, consider the delimiter $r_1 =$ `go`. This delimiter clearly doesn't work; but why not exactly? The reason is that the string `go` satisfies neither of the two properties just described: `go` is not a prefix of any of the `</B> <I>` strings that occur after the countries, and it also is a substring of one of the countries (`Congo`). This simple example illustrates the constraints that the delimiter r_1 must obey.

We can concisely state the constraints on r_1 by introducing some auxiliary terminology. The idea is that we can simplify the specification of the HLRT consistency constraints by assigning "names" to particular "pieces" of the input page. Figure 4.2[2] illustrates that the example page begins with a *head*; each tuple is composed of a set of attribute *values*; values within a single tuple are separated by the *intra-tuple* separators; tuples are separated from one another by the *inter-tuple* separators; finally, the page ends with a *tail*. Note that these various "pieces" constitute a partition of the page; exactly how a page is partitioned depends on its label.

Using this terminology, we can say that `</B>` is a satisfactory value for r_1 because `</B>` is a prefix of the intra-tuple separators between the first and second attribute (labeled $S_{m,1}$ in Figure 4.2), yet `</B>` does not occur in any first attribute's instances (labeled $A_{m,1}$).

The r_1 constraint just described involves only the intra-tuple separators and the

[2] In the figure, the symbol "⇓" indicates a carriage return character; see Appendix C.

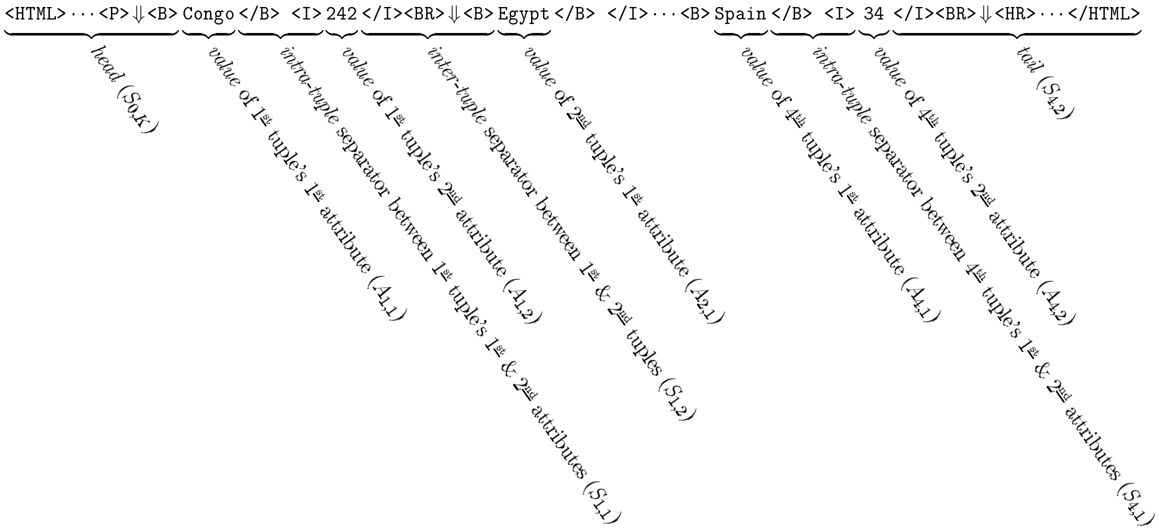

<HTML>···<P>⇓<B>Congo</B> <I>242</I><BR>⇓<B>Egypt</B> </I>···<B>Spain</B> <I> 34 </I><BR>⇓<HR>···</HTML>
head ($S_{0,K}$)
value of 1st tuple's 1st attribute ($A_{1,1}$)
intra-tuple separator between 1st tuple's 1st & 2nd attributes ($S_{1,1}$)
value of 1st tuple's 2nd attribute ($A_{1,2}$)
inter-tuple separator between 1st & 2nd tuples ($S_{1,2}$)
value of 2nd tuple's 1st attribute ($A_{2,1}$)
value of 4th tuple's 1st attribute ($A_{4,1}$)
intra-tuple separator between 4th tuple's 1st & 2nd attributes ($S_{4,1}$)
value of 4th tuple's 2nd attribute ($A_{4,2}$)
tail ($S_{4,2}$)

Figure 4.2: *A label partitions a page into the attribute values, the head, the tail, and the inter- and intra-tuple separators. (For brevity, parts of the page are omitted.)*

attribute values. As we'll see, the other parts of the page's partition are required for the full specification of the HLRT consistency constraints.

The definition of $\mathcal{C}_{\mathrm{HLRT}}$. Definition 4.3 below states the predicate $\mathcal{C}_{\mathrm{HLRT}}$. As we'll see in Section 4.3.2, computing $\mathsf{Generalize}_{\mathrm{HLRT}}(\mathcal{E})$ is a matter of finding a wrapper w such that $\mathcal{C}_{\mathrm{HLRT}}(w, \langle P, L\rangle)$ for every $\langle P, L\rangle \in \mathcal{E}$.

Definition 4.3 (HLRT consistency constraints) *Let* $w = \langle h, t, \ell_1, r_1, \ldots, \ell_K, r_K\rangle$ *be an* HLRT *wrapper, and* $\langle P, L\rangle$ *be an example.* w *and* $\langle P, L\rangle$ *satisfy the* HLRT consistency constraints*—written* $\mathcal{C}_{\mathrm{HLRT}}(w, \langle P, L\rangle)$*—if and only if the predicates* **C1**–**C3** *defined in Figure 4.3 hold:*

$$\begin{array}{rcl} \mathcal{C}_{\mathrm{HLRT}}(w, \langle P, L\rangle) & \iff & \displaystyle\bigwedge_{1\le k\le K} \mathbf{C1}(r_k, \langle P, L\rangle) \\ & \wedge & \displaystyle\bigwedge_{1< k\le K} \mathbf{C2}(\ell_k, \langle P, L\rangle) \\ & \wedge & \mathbf{C3}(h, t, \ell_1, \langle P, L\rangle). \end{array}$$

C1: constraints on the r_k. Every r_k must **(i)** be a prefix of the subsequent intra-tuple separators; and **(ii)** must not occur within any of the corresponding attribute values:

$$\begin{array}{lllll} \mathbf{C1}(r_k, \langle P, L\rangle) & \iff & \forall_{1\le m\le M} & S_{m,k}/r_k = S_{m,k} & \textbf{(i)} \\ & & \wedge & A_{m,k}/r_k = \Diamond. & \textbf{(ii)} \end{array}$$

C2: constraints on the ℓ_k. Every ℓ_k (except ℓ_1) must be a proper suffix of the preceding intra-tuple separators:

$$\mathbf{C2}(\ell_k, \langle P, L\rangle) \iff \forall_{1\le m\le M}\ S_{m,k-1}/\ell_k = \ell_k.$$

C3: constraints on h, t, and ℓ_1. **(i)** ℓ_1 must be a proper suffix of the substring of the page's head following h; **(ii)** t must not occur following h but before ℓ_1 in the page's head; **(iii)** ℓ_1 must not precede t in the page's tail; **(iv)** ℓ_1 must be a proper suffix of each of the inter-tuple separators; and **(v)** t must never precede ℓ_1 in any inter-tuple separator:

$$\begin{array}{llll} \mathbf{C3}(h, t, \ell_1, \langle P, L\rangle) & \iff & (S_{0,K}/h)/\ell_1 = \ell_1 & \textbf{(i)} \\ & \wedge & |(P/h)/t| > |(P/h)/\ell_1| & \textbf{(ii)} \\ & \wedge & |S_{M,K}/\ell_1| > |S_{M,K}/t| & \textbf{(iii)} \\ & \wedge & \forall_{1\le m<M}\ \Big(S_{m,K}/\ell_1 = \ell_1 & \textbf{(iv)} \\ & \wedge & \qquad\qquad |S^*_{m,K}/t| > |S^*_{m,K}/\ell_1|\Big). & \textbf{(v)} \end{array}$$

Figure 4.3: *The* HLRT *consistency constraint $\mathcal{C}_{\mathrm{HLRT}}$ is defined in terms of three predicates* **C1**–**C3**.

The notation $\mathcal{C}_{\mathrm{HLRT}}(w, \mathcal{E})$ is a shorthand for $\forall_{\langle P,L\rangle\in\mathcal{E}}\ \mathcal{C}_{\mathrm{HLRT}}(w, \langle P, L\rangle)$.

Figure 4.3 lists the three predicates **C1**–**C3** in terms of which $\mathcal{C}_{\mathrm{HLRT}}$ is defined. **C1**–**C3** are a convenient way to decompose the predicate $\mathcal{C}_{\mathrm{HLRT}}$ into more easily digested parts. Each predicate governs a particular aspect of consistency. Specifically, **C1** ensures that each delimiter r_k is correct; **C2** governs the ℓ_k (for $k > 1$); and **C3** that h, t, and ℓ_1 are correct. (As we'll see in Section 4.4, the fact that **C1**–**C3** each constrain different HLRT components is the key to designing an efficient $\mathsf{Generalize}_{\mathrm{HLRT}}$ algorithm.)

In the remainder of this section we describe the predicates **C1**–**C3**. The casual reader might prefer to skip directly to Section 4.3.2 on page 52.

String algebra. The specification of **C1**–**C3** in Figure 4.3 makes use of the string operator "/", defined in Appendix C.

The partition variables $A_{m,k}$ and $S_{m,k}$. Figure 4.3 refers to the variables $S_{m,k}$ and $A_{m,k}$. Earlier in this section, Figure 4.2 illustrated how a page is partitioned by its label. The variables $S_{m,k}$ and $A_{m,k}$ occur in the definitions of predicates **C1**–**C3** in order to precisely refer to the parts of this partition.

For example, we saw that the value of r_1 is constrained by the values of the first attribute (referred to with the variables $A_{m,1}$, for each $1 \leq m \leq M$) as well as the text occurring between these attributes values and the next (the $S_{m,1}$). From Figure 4.3, we can recognize this constraint on r_1 as the predicate **C1**.

More formally, suppose P's label L indicates that P contains $M = |L|$ tuples (each consisting of K attributes):

$$L = \left\{ \begin{array}{c} \Big\langle \langle b_{1,1}, e_{1,1}\rangle, \ldots, \langle b_{1,k}, e_{1,k}\rangle, \ldots, \langle b_{1,K}, e_{1,K}\rangle \Big\rangle \\ \vdots \\ \Big\langle \langle b_{m,1}, e_{m,1}\rangle, \ldots, \langle b_{m,k}, e_{m,k}\rangle, \ldots, \langle b_{m,K}, e_{m,K}\rangle \Big\rangle \\ \vdots \\ \Big\langle \langle b_{M,1}, e_{M,1}\rangle, \ldots, \langle b_{M,k}, e_{M,k}\rangle, \ldots, \langle b_{M,K}, e_{M,K}\rangle \Big\rangle \end{array} \right\}.$$

Observe that P is partitioned by L into $2MK+1$ substrings:

$$P \quad = \quad \begin{array}{c} S_{0,K}\ A_{1,1}\ \ S_{1,1}\ \ A_{1,2}\ \ S_{1,2} \cdots S_{1,K\text{–}1}\ \ A_{1,K}\ S_{1,K} \\ \vdots \\ A_{m,1}\, S_{m,1}\, A_{m,2}\, S_{m,2} \cdots S_{m,K\text{–}1}\, A_{m,K}\, S_{m,K} \\ \vdots \\ A_{M,1}\, S_{M,1}\, A_{M,2}\, S_{M,2} \cdots S_{M,K\text{–}1}\, A_{M,K}\, S_{M,K}. \end{array} \qquad (4.1)$$

This partition of P is defined in terms of MK values of $A_{m,k}$ and $MK+1$ values of $S_{m,k}$. These variables are defined as follows:

- The $A_{m,k}$ are the values of each *attribute* in each of page P's tuples. Specifically, $A_{m,k}$ is the value of the $k^{\underline{\text{th}}}$ attribute of the $m^{\underline{\text{th}}}$ tuple on page P. In terms of the indices in label L:[3]

  $$A_{m,k} = P\left[b_{m,k}, e_{m,k}\right].$$

  The $A_{m,k}$, of course, are simply the text fragments to be extracted from page P.

- The $S_{m,k}$ are the *separators* between these attribute values. There are four kinds of separators:
  - Page P's *head*, denoted $S_{0,K}$, is the substring of the page prior to the first attribute of the first tuple:

    $$S_{0,K} = P\left[0, b_{1,1}\right].$$

  - Page P's *tail*, denoted by $S_{M,K}$, is the substring of the page following the last attribute of the last tuple:

    $$S_{M,K} = P\left[e_{M,K}, |P|\right].$$

  - The *intra-tuple separators* separate attributes within a single tuple. Specifically, $S_{m,k}$ (for all $1 \leq k < K$) is the separator between the $k^{\underline{\text{th}}}$ and $(k+1)^{\underline{\text{st}}}$ attribute of the $m^{\underline{\text{th}}}$ tuple in page P:

    $$S_{m,k} = P\left[e_{m,k}, b_{m,k+1}\right].$$

  - The *inter-tuple separators* separate consecutive tuples. Specifically, $S_{m,K}$ is the separator between the $m^{\underline{\text{th}}}$ and $(m+1)^{\underline{\text{st}}}$ tuple on page P, for $1 \leq m < M$:

    $$S_{m,K} = P\left[e_{m,K}, b_{m+1,1}\right].$$

[3] Recall from Appendix C that the notation $s[b,e]$ is the substring of s from position b to position e.

> Note that the subscripts on the head ($S_{0,K}$) and tail ($S_{M,K}$) generalize the inter-tuple separators: the head "separates" the "zeroth" and first tuples, and the tail "separates" the last and "post-ultimate" tuples.

Note that the $A_{m,k}$ and $S_{m,k}$ do in fact partition P, because collectively they cover the entire string P.

Finally, the notation $S^*_{m,k}$ refers to the concatenation of $S_{m,k}$ with *all subsequent* partition elements. More precisely,

$$S^*_{m,k} = S_{m,k}A_{m,k+1}S_{m,k+1} \cdots A_{m,K}S_{m,K}A_{m+1,1}S_{m+1,1} \cdots A_{M,K}S_{M,K}.$$

For example, under this notation, page P can be written as $P = S^*_{0,K}$, because a page consists of its head ($S_{0,K}$) followed by the rest of the page. Note also that $S_{M,K} = S^*_{M,K}$, because nothing follows a page's tail.

Before proceeding, let us clarify that the notation $A_{m,k}$ and $S_{m,k}$ obscures an important point: the value of each partition variable obviously depends on the page being partitioned. For example, given a set of pages $P_1, \ldots, P_N$, it is ambiguous to which page the variables $A_{m,k}$ or $S_{m,k}$ refer. Rather than further complicate the notation (*e.g.*, by noting the relationship with a superscript: $A^n_{m,k}$) we will always explicitly mention the page under consideration.

Explanation of C3(iv-v). The predicates **C1**–**C3** are tersely presented in Table 4.3. The details are required only for the proofs in Appendix B. The casual reader need not comprehend **C1**–**C3** at such depth; the English-language descriptions given in Figure 4.3 should suffice. Nevertheless, let us now explain part of the notation in order to motivate and explain **C1**–**C3**. Predicate **C3** is relatively complex, and so exploring just **C3(iv-v)** in detail will illuminate the rest.

Predicate **C3(iv-v)** ensures that ℓ_1 and t are satisfactory with regard to a page's body. The '$\forall_{1 \leq m < M}$' quantifier states that **C3(iv-v)** must hold for each tuple m except the first and last (which are handled by **C3(i–ii)** and **C3(iii)**, respectively).

C3(iv), $S_{m,K}/\ell_1 = \ell_1$, ensures that ℓ_1 is a proper prefix of every $S_{m,K}$. To see this, note that for any strings s and s', $s/s' = s'$ iff s' is a proper suffix of s.

C3(v), $|S^*_{m,K}/t| > |S^*_{m,K}/\ell_1|$, ensures that the tail delimiter t must always occur after ℓ_1 in the $m^{\underline{\text{th}}}$ inter-tuple separator. To see this, note that $|s/s'|$ is the position of s' in s; therefore $|s/s'| > |s/s''|$ iff s' occurs after s'' in s.

$\mathcal{C}_{\text{HLRT}}$ is correct. We have defined a predicate $\mathcal{C}_{\text{HLRT}}$ and claimed that it exactly captures the conditions under which a wrapper is consistent with a particular example. We now formalize this claim.

> **Theorem 4.1 ($\mathcal{C}_{\text{HLRT}}$ is correct)** *For every* HLRT *wrapper w, page P, and label L,* $\mathcal{C}_{\text{HLRT}}(w, \langle P, L\rangle) \iff \mathsf{ExecHLRT}(w, P) = L$.

See Appendix B for the proof.

Summary. In this section we have defined the HLRT consistency constraints, the conditions under which a wrapper is consistent with a given example. Specifically, we have introduced the predicate $\mathcal{C}_{\text{HLRT}}(w, \langle P, L\rangle)$, which holds if and only if $w(P) = L$—*i.e.*, just in case HLRT wrapper w generates label L for page P.

$\mathcal{C}_{\text{HLRT}}$ is specified in terms of a complicated set of notation. $\mathcal{C}_{\text{HLRT}}$ is defined in terms of a conjunction of more primitive predicates, **C1**–**C3**. In turn, **C1**–**C3** are defined in terms of the variables $A_{m,k}$ and $S_{m,k}$, which indicate the way that L partitions P into a set of attribute values and the separators occurring between them.

4.3.2 Generalize$_{\text{HLRT}}$

We introduced the HLRT consistency constraint predicate $\mathcal{C}_{\text{HLRT}}$ because it provides a straightfoward way to describe the Generalize$_{\text{HLRT}}$ algorithm: Generalize$_{\text{HLRT}}$ is a special-purpose constraint-satisfaction engine that takes as input a set $\mathcal{E} = \langle\langle P_1, L_1\rangle, \ldots, \langle P_N, L_N\rangle\rangle$, and that solves problems of the following form:

```
Generalize_HLRT(examples $\mathcal{E} = \{\langle P_1, L_1\rangle, \ldots, \langle P_N, L_N\rangle\}$)
  for $r_1 \leftarrow$ each prefix of $P_1$'s intra-tuple separator for $S_{1,1}$          4.4(a)
    $\ddots$
      for $r_K \leftarrow$ each prefix of $P_1$'s intra-tuple separator $S_{1,K}$          4.4(b)
        for $\ell_1 \leftarrow$ each suffix of $P_1$'s head $S_{0,K}$                       4.4(c)
          for $\ell_2 \leftarrow$ each suffix of $P_1$'s intra-tuple separator $S_{1,1}$    4.4(d)
            $\ddots$
              for $\ell_K \leftarrow$ each suffix of $P_1$'s intra-tuple separator $S_{1,K-1}$   4.4(e)
                for $h \leftarrow$ each substring of $P_1$'s head $S_{0,K}$                 4.4(f)
                  for $t \leftarrow$ each substring of $P_1$'s tail $S_{M,K}$               4.4(g)
                    $w \leftarrow \langle h, t, \ell_1, r_1, \ldots, \ell_K, r_K\rangle$    4.4(h)
                    if $\mathcal{C}_{\mathrm{HLRT}}(w, \mathcal{E})$, then                  4.4(i)
                      return $w$                                                            4.4(j)
```

Figure 4.4: *The* Generalize$_{\mathrm{HLRT}}$ *algorithm.*

Variables: h, t, ℓ_1, r_1, $\ldots$, ℓ_K, r_K
Domains: each variable can be an arbitrary character string
Constraints: $\mathcal{C}_{\mathrm{HLRT}}(w, \mathcal{E})$, where $w = \langle h, t, \ell_1, r_1, \ldots, \ell_K, r_K\rangle$

The algorithm. Figure 4.4 presents Generalize$_{\mathrm{HLRT}}$, an algorithm that solves constraint-satisfaction problems of this form.

Generalize$_{\mathrm{HLRT}}$ is a simple generate-and-test algorithm. Given input $\mathcal{E}$, Generalize$_{\mathrm{HLRT}}$ operates by searching the space of HLRT wrappers for a wrapper w that satisfies $\mathcal{C}_{\mathrm{HLRT}}$ with respect to each example. Generalize$_{\mathrm{HLRT}}$ employs a depth-first search strategy: the algorithm considers all candidate values for r_1; for each such r_1, it then considers all candidate values for r_2; for each such r_1 and r_2, it then considers all candidate values for r_3; and so on. The result is a loop control structure nested $2K + 2$ levels deep.

What candidates values should be considered for each of the $2K + 2$ HLRT components? An implementation of Generalize$_{\mathrm{HLRT}}$ that does not restrict these candidate sets is infeasible, because the HLRT space is infinite and thus a depth-first search might never terminate.

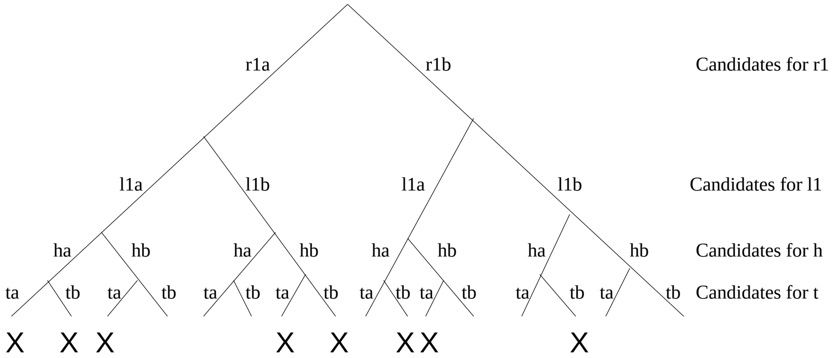


Figure 4.5: *The space searched by* Generalize$_{\text{HLRT}}$*, for a very simple example.*

However, on the basis of just a single example, the number of candidates for each of the $2K + 2$ components becomes finite. For example, the head delimiter h must be a substring of the head of each page P_n. Thus line 4.4(f) uses as candidates for h, not all possible strings, but rather only those that are subtrings of page P_1's head $S_{0,K}$. Similar constraints apply to each HLRT component.

Wrapper induction as search. In Section 4.4 we will perform a detailed complexity analysis of Generalize$_{\text{HLRT}}$; the bottom line is that Generalize$_{\text{HLRT}}$ examines a finite search space. This means that we can describe the algorithm in very simple terms: Generalize$_{\text{HLRT}}$ searches the finite space of potentially-consistent HLRT wrappers in a depth-first fashion, stopping when it encounters a wrapper w that obeys $\mathcal{C}_{\text{HLRT}}$.

Figure 4.5 illustrates the space searched by Generalize$_{\text{HLRT}}$, for a very simple example in which there is just $K = 1$ attribute per tuple, and where there are exactly two candidates for each of the $2K + 2 = 4$ wrapper components. In the Figure, the two candidates for h are denoted "ha" and "hb". Similarly, for t the candidates are "ta" and "tb"; for ℓ_1, "l1a" and "l1b"; and for r_1, "r1a" and "r1b". Figure 4.5 shows the space as a binary tree. In general, the tree is not binary: the branching factor at each level captures the number of candidates considered by the corresponding line of

Generalize$_{\text{HLRT}}$ (in this case, line 4.4(a) for the first level, 4.4(c) for the second, 4.4(f) for the third, and 4.4(g) for the fourth).

The search space is a tree. Leaves represent wrappers, and interior nodes indicate the selection of a candidate for a specific wrapper component. Some of the leaves are labeled 'X', indicating that the corresponding wrapper does not satify $\mathcal{C}_{\text{HLRT}}$ (these leaves are chosen arbitrarily merely for the sake of illustration). For example, the right-most leaf in the tree represents the wrapper $\langle \text{hb}, \text{tb}, \text{l1b}, \text{r1b} \rangle$.

Using these ideas, we can now succinctly describe our algorithm: Generalize$_{\text{HLRT}}$ does an exhaustive depth-first search of the tree for a leaf not labeled 'X'. Since the out-degree of each interior node and the tree's depth are finite, such a search process is guaranteed to terminate.

4.3.3 Example

Before describing the formal properties of Generalize$_{\text{HLRT}}$, we illustrate how the algorithm operates using the country/code example.

Suppose Generalize$_{\text{HLRT}}$ is given just a single example, the HTML page in Figure 2.1(c) (call it page P_{cc}) together with its label, Equation 2.2 (call it L_{cc}). The following "partial evaluation" of Generalize$_{\text{HLRT}}$ illustrates how this example is processed:

```
Generalize_HLRT(example $\{\langle P_{cc}, L_{cc} \rangle\}$)
  for $r_1 \leftarrow$ each prefix of </B> <I>                                    †
    for $r_2 \leftarrow$ each prefix of </I>⇓<B>
      for $\ell_1 \leftarrow$ each suffix of <HTML><TITLE>···<P>⇓<B>
        for $\ell_2 \leftarrow$ each suffix of </B> <I>
          for $h \leftarrow$ each substring of <HTML><TITLE>···<P>⇓<B>
            for $t \leftarrow$ each substring of </I><BR>⇓<HR>···</HTML>
              $w \leftarrow \langle h, t, \ell_1, r_1, \ell_2, r_2 \rangle$
              if $\mathcal{C}_{HLRT}(w, \langle P, L \rangle)$, then return $w$
```

Consider the first line of this code fragment, marked "†". As specified in Figure 4.4, this line lists the candidates for HLRT component r_1: the prefixes of </B> <I>. To see that these are the candidates, note that `</B> <I>` is the string occuring after the first attribute of the first tuple in the example $\langle P_{\mathrm{cc}}, L_{\mathrm{cc}}\rangle$. That is, $S_{1,1}$ = `</B> <I>` for page P_{cc}. Similarly, the candidates for r_2 are the prefixes of `</I>`⇓`<B>` because $S_{1,2}$ = `</I>`⇓`<B>`, and so forth for the remaining HLRT components.

$\mathsf{Generalize}_{\mathrm{HLRT}}$ operates by iterating over all possible combinations of the candidates. Each such combination corresponds to a wrapper $\langle h, t, \ell_1, r_1, \ell_2, r_2\rangle$. Each wrapper is tested to see whether it satisfies $\mathcal{C}_{\mathrm{HLRT}}$; if so, $\mathsf{Generalize}_{\mathrm{HLRT}}$ terminates and the wrapper is returned. In this particular example, eventually the wrapper ⟨`<P>`, `<HR>`, `<B>`, `</B>`, `<I>`, `</I>`⟩ is encountered, and $\mathsf{Generalize}_{\mathrm{HLRT}}$ halts.[4]

4.3.4 Formal properties

As described in Definition 4.2, we want the function $\mathsf{Generalize}_{\mathrm{HLRT}}$ to be consistent—*i.e.*, $\mathsf{Generalize}_{\mathrm{HLRT}}$ should output only wrappers w that are correct for the input examples. In this section, we formalize this claim.

Theorem 4.2 $\mathsf{Generalize}_{\mathrm{HLRT}}$ *is consistent.*

Proof of Theorem 4.2: From lines 4.4(h–j), we know that if $\mathsf{Generalize}_{\mathrm{HLRT}}$ returns a wrapper w, then w will satisfy $\mathcal{C}_{\mathrm{HLRT}}$ for every example. Theorem 4.1 states $\mathcal{C}_{\mathrm{HLRT}}(w, \langle P, L\rangle) \;\Rightarrow\; w(P) = L$. Thus establishing Lemma 4.3 completes the proof.

□ (Proof of Theorem 4.2)

Lemma 4.3 ($\mathsf{Generalize}_{\mathrm{HLRT}}$ is complete) *For any set $\mathcal{E}$ of examples, if there exists an* HLRT *wrapper obeying $\mathcal{C}_{\mathrm{HLRT}}$ for each member of $\mathcal{E}$, then* $\mathsf{Generalize}_{\mathrm{HLRT}}(\mathcal{E})$ *will return such a wrapper.*

[4] Actually, exhaustive enumeration reveals that there are more than 250 million HLRT wrappers that are consistent with $\langle P_{\mathrm{cc}}, L_{\mathrm{cc}}\rangle$; the wrapper ⟨`Some`, ⇓`<HR>`, ⇓`<B>`, `</B> <I>`, `</B> <I>`, `</I>`⇓⟩ is one example. Since we have not specified the order in which the candidates are examined, we can not be certain which wrapper will be returned. These subtle search-control issues are in interesting direction of future research; see Chapter 9.

The proof of Lemma 4.3 essentially involves establishing that $\mathsf{Generalize}_{\mathrm{HLRT}}$ considers enough candidates for each HLRT component. For example, when choosing a value for h, we need to show that satisfactory values for h are always substrings of page P_1's head, and thus $\mathsf{Generalize}_{\mathrm{HLRT}}$ will always find a satisfactory value even though it restricts its search to such substrings. See Appendix B.2 for the details.

4.4 *Efficiency: The* $\mathsf{Generalize}^{*}_{\mathrm{HLRT}}$ *algorithm*

Though very simple and exhibiting nice formal properties, the $\mathsf{Generalize}_{\mathrm{HLRT}}$ algorithm as described in Section 4.3.2 is very inefficient. In this section we analyze the algorithm's computational complexity, and describe several efficiency improvements. The result is the algorithm $\mathsf{Generalize}^{*}_{\mathrm{HLRT}}$, which—like $\mathsf{Generalize}_{\mathrm{HLRT}}$—is consistent, but which is much faster.

4.4.1 Complexity analysis of $\mathsf{Generalize}_{\mathrm{HLRT}}$

What is the computational complexity of $\mathsf{Generalize}_{\mathrm{HLRT}}$? The algorithm consists of $2K + 2$ nested loop. We can bound the total running time of $\mathsf{Generalize}_{\mathrm{HLRT}}$ by multiplying the number of times line 4.4(i) is executed (*i.e.*, the total number of iterations of the $2K + 2$ nested loops) by the time to execute line 4.4(i) once.

How many times does each nested loop iterate? Each substring of page P_1's head is a candidate value for h. How many such substrings are there? Without loss of generality, and to obtain the tightest bound, we can assume that $\mathsf{Generalize}_{\mathrm{HLRT}}$ enumerates the substrings of the *shortest* page's head—*i.e.*, Figure 4.4 assumes that P_1 is the shortest page. Note that page P_1's $|S_{0,K}|$ grows with $|P_1|$ in the worst case. Thus we can use $R = |P_1|$ to bound the number of candidates for h. Specifically, since there are $\frac{R(R-1)}{2}$ substrings of a string of length R, $\mathsf{Generalize}_{\mathrm{HLRT}}$ must consider $\mathrm{O}\left(R^2\right)$ candidates for h. A similar argument applies to the tail delimiter t: $\mathsf{Generalize}_{\mathrm{HLRT}}$ must examine $\mathrm{O}\left(R^2\right)$ candidates for t.

The number of candidates for the r_k and l_k are constrained more tightly. For example, r_1 must be a prefix of each $S_{m,1}$, the intra-tuple separators between the first and second attributes on each page. Thus the candidates for r_1 can be enumerated by simply considering all prefixes of the shortest value of $S_{m,k}$. At this point the analysis proceeds as with the page heads and tails: $|S_{m,k}|$ is at most R, and there are R prefixes of a string of length R, and therefore $\mathsf{Generalize}_{\mathrm{HLRT}}$ must consider $\mathrm{O}(R)$ candidates for r_1. A similiar argument applies to, and the same bound can be derived for, each r_k and ℓ_k.

Multiplying these bounds together, we have that $\mathsf{Generalize}_{\mathrm{HLRT}}$ must execute line 4.4(i) for

$$\mathrm{O}\left(R^2\right) \times \mathrm{O}\left(R^2\right) \times \underbrace{\mathrm{O}(R) \times \cdots \times \mathrm{O}(R)}_{2K \text{ terms}} = \mathrm{O}\left(R^{2K}\right) \tag{4.2}$$

for different wrappers.

How long does it take to evaluate $\mathcal{C}_{\mathrm{HLRT}}$ for a single wrapper—*i.e.*, how long does it take to execute line 4.4(i)? Computing $\mathcal{C}_{\mathrm{HLRT}}$ essentially consists of a sequence of string-search operations (written as 's/s'') over pairs of strings that, in the worst case, each have length R. Using efficient techniques, (*e.g.*, [Knuth et al. 77]), each such operation takes time $\mathrm{O}(R)$. Thus, bounding the time to compute $\mathcal{C}_{\mathrm{HLRT}}$ amounts to multiplying the total number of invocations of the string-search operator by the time per invocation, $\mathrm{O}(R)$.

The total number of such invocations can be computed by summing the number of invocations required by each of the three predicates **C1**–**C3**. We must test each predicate against each of the N examples. Predicates **C1** and **C2** are invoked $\mathrm{O}(K)$ times, once per attribute. Each such evaluation requires examining each of the tuples of each page. We can bound the number of tuples on a single page as $M_{\max} = \max_n M_n$, where $M_n = |L_n|$ is the number of tuples on page P_n. Thus the time to evaluate **C1** and **C2** for a single page is $\mathrm{O}(KM_{\max})$. Evaluating **C3** involves a constant number of string search operations for each tuple across all of the pages. Thus, testing **C3** on a single page involves $\mathrm{O}\left(M_{\max}\right)$ primitive string-search operations. Therefore, the time to test **C1**–**C3** on a single example is $\mathrm{O}\left(KM_{\max}\right) + \mathrm{O}\left(KM_{\max}\right) + \mathrm{O}\left(M_{\max}\right) = \mathrm{O}\left(KM_{\max}\right)$. Therefore, the time to test **C1**–**C3** against all the examples is $\mathrm{O}\left(KM_{\max}\right) \times \mathrm{O}\left(N\right) = \mathrm{O}\left(KM_{\max}N\right)$. Since the time to perform a single string-search operator is $\mathrm{O}\left(R\right)$, the total time to compute $\mathcal{C}_{\mathrm{HLRT}}$ for a single wrapper is

$$\mathrm{O}\left(KM_{\max}N\right) \times \mathrm{O}\left(R\right) = \mathrm{O}\left(KM_{\max}NR\right).$$

Multiplying this bound by the number of wrappers for which $\mathcal{C}_{\mathrm{HLRT}}$ must be evaluated (Equation 4.2) reveals that the total running time of $\mathsf{Generalize}_{\mathrm{HLRT}}$ is

$$\mathrm{O}\left(KM_{\max}NR\right) \times \mathrm{O}\left(R^{2K}\right) = \mathrm{O}\left(M_{\max}NR^{2K+1}\right).$$

To summarize, we have established the following theorem.

> **Theorem 4.4 ($\mathsf{Generalize}_{\mathrm{HLRT}}$ complexity)** *The* $\mathsf{Generalize}_{\mathrm{HLRT}}$ *algorithm runs in time* $O\left(M_{\max}NR^{2K+1}\right)$*, where* K *is the number of attributes per tuple,* $N = |\mathcal{E}|$ *is the number of examples,* $M_{\max} = \max_n M_n$ *is the maximum number of tuples on any single page,* $M_n = |L_n|$ *is the number of tuples on page* P_n*, and* $R = \min_n |P_n|$ *is the length of the shortest example page.*

4.4.2 $\mathsf{Generalize}^{*}_{\mathrm{HLRT}}$

As the preceding complexity analysis indicates, $\mathsf{Generalize}_{\mathrm{HLRT}}$ is a relatively naïve algorithm for solving the constraint-satisfaction problem of finding a wrapper consistent with a set of examples. In this section, we improve the performance of $\mathsf{Generalize}_{\mathrm{HLRT}}$ by describing several optimizations.

The basic idea is that $\mathsf{Generalize}_{\mathrm{HLRT}}$ can be improved by decomposing the problem of finding the entire wrapper w into the problem of finding each of w's $2K+2$ component delimiters (h, t, ℓ_1, r_1, *etc.*). The key insight is that (for the most part) the predicates **C1**–**C3** do not interact, and thus we can find each delimiter with (in most cases) no regard for the others.

For example, consider the variable r_2. Inspecting predicate **C1** for $k = 2$,

$$\mathbf{C1}(r_2, P, L) \iff \forall_{1 \leq m \leq M} \left(S_{m,k}/r_2 = S_{m,k}\right) \wedge \left(A_{m,k}/r_2 = \Diamond\right),$$

it is apparent that constraint **C1** governs only r_2. Furthermore, the other predicates (**C2**–**3**) do not mention r_2. Therefore, we can assign a value to r_2 without regard to assignments to *any* of the other $2K - 1$ variables. This observation suggests that the complexity of finding r_2 is considerably simpler than indicated by the bound derived for $\mathsf{Generalize}_{\mathrm{HLRT}}$.

To generalize, rather than solving the overall constraint-satisfaction problem of finding a wrapper w obeying $\mathcal{C}_{\mathrm{HLRT}}$, we can instead solve for the $2K + 2$ components of w as follows.

- Each variable r_k (for $1 \leq k \leq K$) is independent of the remaining $2K + 1$ variables, since each r_k is governed only by predicate **C1**.
- Similarly, each variable ℓ_k (for $1 < k \leq K$) is independent of the remaining $2K + 1$ variables, since each ℓ_k is governed only by predicate **C2**.
- The remaining three variables, h, t, and ℓ_1, mutually constrain each other, but assignments to these three variables are independent of assignments to the other $2K - 1$ variables. To see this, note that predicate **C3** mentions h, t, and ℓ_1, but neither of the other two predicates mention either of these variables.

The algorithm. Figure 4.6 lists $\mathsf{Generalize}^*_{\mathrm{HLRT}}$, an improved version of $\mathsf{Generalize}_{\mathrm{HLRT}}$ that makes use of these improvements. $\mathsf{Generalize}^*_{\mathrm{HLRT}}$ operates by solving for each r_k, then for each ℓ_k ($k > 1$), and finally for the three values ℓ_1, h and t. In a nutshell, $\mathsf{Generalize}^*_{\mathrm{HLRT}}$ is much faster than $\mathsf{Generalize}_{\mathrm{HLRT}}$ because $\mathsf{Generalize}^*_{\mathrm{HLRT}}$'s iteration constructs are in series rather than nested.

Fast wrapper induction as search. Recall Figure 4.5, which illustrates the space searched by the $\mathsf{Generalize}_{\mathrm{HLRT}}$ algorithm. Notice that the choice of r_1 is represented

```
$\underline{\mathsf{Generalize}^*_{\mathrm{HLRT}}}$(examples $\mathcal{E} = \{\langle P_1, L_1\rangle, \ldots, \langle P_N, L_N\rangle\}$)
    for each $1 \leq k \leq K$                                                          4.6(a)
      for $r_k \leftarrow$ each prefix of $P_1$'s intra-tuple separator for $S_{1,k}$           4.6(b)
        accept $r_k$ if **C1** holds of $r_k$ and every $\langle P_n, L_n\rangle \in \mathcal{E}$          4.6(c)
    for each $1 < k \leq K$                                                          4.6(d)
      for $\ell_k \leftarrow$ each suffix of $P_1$'s intra-tuple separator $S_{1,k-1}$           4.6(e)
        accept $\ell_k$ if **C2** holds of $\ell_k$ and every $\langle P_n, L_n\rangle \in \mathcal{E}$          4.6(f)
    for $\ell_1 \leftarrow$ each suffix of $P_1$'s head $S_{0,K}$                               4.6(g)
      for $h \leftarrow$ each substring of $P_1$'s head $S_{0,K}$                            4.6(h)
        for $t \leftarrow$ each substring of $P_1$'s tail $S_{M,K}$                          4.6(i)
          accept $\ell_1$, $h$, and $t$ if **C3** holds of $\ell_1$, $h$, $t$ and every $\langle P_n, L_n\rangle \in \mathcal{E}$   4.6(j)
    return $\langle h, t, \ell_1, r_1, \ldots, \ell_K, r_K\rangle$                              4.6(k)
```

Figure 4.6: *The* $\mathsf{Generalize}^*_{\mathrm{HLRT}}$ *algorithm is an improved version of* $\mathsf{Generalize}_{\mathrm{HLRT}}$ *(Figure 4.4).*

as the tree's root. Similarly, the choice for ℓ_1 is represented by the first layer of nodes; the choice of h, the second layer; and the choice for t, the third layer. This ordering derives directly from the fact that in the original $\mathsf{Generalize}_{\mathrm{HLRT}}$ (Figure 4.4), the outer-most loop iterates over candidates for r_1, the next loop iterates over candidates for r_2, and so forth, ending with the inner-most loop iterating over candidates for t.

Given this observation, we can describe the improved $\mathsf{Generalize}^*_{\mathrm{HLRT}}$ algorithm in very simple terms. Like $\mathsf{Generalize}_{\mathrm{HLRT}}$, $\mathsf{Generalize}^*_{\mathrm{HLRT}}$ searches the tree in Figure 4.5 in a depth-first fashion. However, $\mathsf{Generalize}^*_{\mathrm{HLRT}}$ backtracks *only* over the bottom three node layers—*i.e.*, the nodes representing choices for ℓ_1, h and t. The algorithm does *not* backtrack over the nodes from the root down to the fourth layer from the bottom—*i.e.*, the nodes representing choices for $r_1, \ldots, r_K, \ell_2, \ldots, \ell_K$.

$\mathsf{Generalize}^*_{\mathrm{HLRT}}$'s consistency proof (presented next) mainly involves showing that this greedy search never fails to find a consistent wrapper if one exists. The basic idea is that while $\mathsf{Generalize}_{\mathrm{HLRT}}$ uses the "global" evaluation function $\mathcal{C}_{\mathrm{HLRT}}$ to determine whether a leaf is labeled 'X' (*i.e.*, the corresponding wrapper is consistent), $\mathsf{Generalize}^*_{\mathrm{HLRT}}$ uses the "local" evaluation functions **C1**–**C3**.

4.4.3 Formal properties

In attempting to improve $\mathsf{Generalize}_{\mathrm{HLRT}}$, have we sacrificed any formal properties? First of all, note that if there are several HLRT wrappers consistent with a given set of examples, then the two algorithms might return different wrappers. So in general we can not say that $\mathsf{Generalize}_{\mathrm{HLRT}}(\mathcal{E}) = \mathsf{Generalize}^{*}_{\mathrm{HLRT}}(\mathcal{E})$ for a set of examples $\mathcal{E}$. However, like $\mathsf{Generalize}_{\mathrm{HLRT}}$ (Theorem 4.2), $\mathsf{Generalize}^{*}_{\mathrm{HLRT}}$ is consistent:

> **Theorem 4.5** $\mathsf{Generalize}^{*}_{\mathrm{HLRT}}$ *is consistent.*

See Appendix B.3 for the proof.

4.4.4 Complexity analysis of $\mathsf{Generalize}^{*}_{\mathrm{HLRT}}$

What is the complexity of $\mathsf{Generalize}^{*}_{\mathrm{HLRT}}$? To perform this analysis, we determine a bound on the running time of each of the three outer loops, lines 4.6(a–c), 4.6(d–f) and 4.6(g–j). As before, we assume that each primitive string search operation takes time $\mathrm{O}(R)$, where $R = \min_n |P_n|$ is the length of the shortest example page.

The first loop (lines 4.6(a–c)) iterates K times, and each iteration requires $\mathrm{O}(M_{\mathrm{max}}N)$ string-search operations (where K is the number of attributes per tuple, N is the number of examples, and M_{tot} is the total number of tuples in the examples). Thus the total running time of lines 4.6(a–c) is $\mathrm{O}(K) \times \mathrm{O}(M_{\mathrm{max}}N) \times \mathrm{O}(R) = \mathrm{O}(KM_{\mathrm{max}}NR)$. A similar analysis reveals that lines 4.6(d–f) also run in time $\mathrm{O}(KM_{\mathrm{max}}NR)$.

The final loop construct, lines 4.6(g–j), is more interesting. This triply-nested loop construct considers all combinations of values for ℓ_1, h and t. As before, there are $\mathrm{O}(R)$ candidates for ℓ_1 and $\mathrm{O}\left(R^2\right)$ each for h and t, for a total of $\mathrm{O}\left(R^5\right)$ evaluations of predicate **C3**. Each such evaluation involves $\mathrm{O}(M_{\mathrm{max}}N)$ invocations of the string-search operator. Thus, lines 4.6(g–j) run in time

$$\mathrm{O}\left(R^5\right) \times \mathrm{O}(M_{\mathrm{max}}N) \times \mathrm{O}(R) = \mathrm{O}\left(M_{\mathrm{max}}NR^6\right). \tag{4.3}$$

Summing these three bounds, we have that $\mathsf{Generalize}^*_{\mathrm{HLRT}}$ is bounded by

$$\mathrm{O}\left(KM_{\max}NR\right) + \mathrm{O}\left(KM_{\max}NR\right) + \mathrm{O}\left(M_{\max}NR^6\right) = \mathrm{O}\left(KM_{\max}NR^6\right),$$

which provides a proof of the following theorem.

> **Theorem 4.6 ($\mathsf{Generalize}^*_{\mathrm{HLRT}}$ complexity)** *The* $\mathsf{Generalize}^*_{\mathrm{HLRT}}$ *algorithm runs in time* $O\left(KM_{\max}NR^6\right)$.

$\mathsf{Generalize}^*_{\mathrm{HLRT}}$ is therefore significantly more efficient than $\mathsf{Generalize}_{\mathrm{HLRT}}$, which runs in time $\mathrm{O}\left(M_{\max}NR^{2K+1}\right)$ (Theorem 4.4). This improvement derives from the observation that each variable r_k and ℓ_k (except ℓ_1) can be learned independently of the remaining three variables.

Since $\mathsf{Generalize}^*_{\mathrm{HLRT}}$ is demonstrably superior to $\mathsf{Generalize}_{\mathrm{HLRT}}$, we will henceforth drop the '$*$' annotation; unless explicitly noted, the notation $\mathsf{Generalize}_{\mathrm{HLRT}}$ will refer to the more efficient generalization algorithm.

4.5 Heuristic complexity analysis

The $\mathrm{O}\left(KM_{\max}NR^6\right)$ worst-case running time bound derived for $\mathsf{Generalize}^*_{\mathrm{HLRT}}$ represents a substantial improvement over the $\mathrm{O}\left(M_{\max}NR^{2K+1}\right)$ bound for $\mathsf{Generalize}_{\mathrm{HLRT}}$. Unfortunately, even if R were relatively small, a degree-six polynomial is unlikely to be useful in practice. Moreover, R is *not* relatively small: for the Internet resources examined in Chapter 7, R ranges from 1,000–30,000 characters. Fortunately, it turns out that the worst-case assumptions are extremely pessimistic. By being only slightly more optimistic, we can obtain a much better bound.

We make the worst-case analysis less pessimistic on the basis of the following observation. Recall that for each wrapper component, we try all candidate substrings of some particular partition fragments of the first page P_1 ($S_{0,K}$ for h, $S_{1,4}$ for r_4, *etc.*—see lines 4.6(b,e,g–i) of the $\mathsf{Generalize}^*_{\mathrm{HLRT}}$ algorithm for details). Our complexity bound was derived in part by counting the number of such candidates for each HLRT component. And these counts depend on the length of these partition fragments. So far, we assumed that these lengths are bounded by the length of the shortest page, $R = \min_n |P_n|$. But in practice, any individual partition fragment constitutes only a small fraction of page P_n's total length: $|S_{m,k}| \ll |P_n|$ and $|A_{m,k}| \ll |P_n|$, for every n, k and m.

Why should the individual $A_{m,k}$ and $S_{m,k}$ be much shorter than the pages themselves? First of all, since the $A_{m,k}$ and $S_{m,k}$ form a partition of the page, on average the partition elements simply can't have length approximately R. And from a more empirical perspective, we have observed that the bulk of an information resource's query responses consist of formatting commands, advertisements, and other irrelevant text; the extracted content of the page is usually quite small.

The idea of the heuristic complexity analysis is to formalize and exploit this intuition. Specifically, our heuristic analysis rests on the following assumption:

> **Assumption 4.1 (Short page fragments)** *On average, page P_n's partition elements $S_{m,k}$ and $A_{m,k}$ all have length approximately $\sqrt[3]{R}$ (rather than R).*

(The specific function $\sqrt[3]{R}$ was chosen to simplify the following analysis. However, as we will see in Section 7.5, this function is very close to what is observed empirically at actual Internet sites.)

What does Assumption 4.1 buy us? In the remainder of this Section, we perform a "heuristic-case" analysis of $\mathsf{Generalize}_{\text{HLRT}}$'s running time. Like an average-case analysis, this heuristic-case analysis provides a better estimate of the running-time of our algorithm. While an average-case analysis assumes a specific probability distribution over the algorithm's inputs, our heuristic-case analysis relies on particular properties of these inputs (namely, our analysis rests on Assumption 4.1).

As indicated in Equation 4.3, the triply-nested loop structure (lines 4.6(g–j)), which runs in time $\mathrm{O}\left(M_{\max} N R^6\right)$, dominates $\mathsf{Generalize}_{\text{HLRT}}$'s running time:

```
for $\ell_1 \leftarrow$ each suffix of $P_1$'s head $S_{0,K}$                                  4.6(g)
  for $h \leftarrow$ each substring of $P_1$'s head $S_{0,K}$                                  4.6(h)
    for $t \leftarrow$ each substring of $P_1$'s tail $S_{M,K}$                                 4.6(i)
      accept $\ell_1$, $h$, and $t$ if **C3** holds of $\ell_1$, $h$, $t$ and every $\langle P_n, L_n\rangle \in \mathcal{E}$      4.6(j)
```

Walking through lines 4.6(g–j), we can compute a new, tighter bound on the basis of Assumption 4.1, as follows. Since there are $\mathrm{O}\left(\sqrt[3]{R}\right)$ suffixes of a string of length $\sqrt[3]{R}$, line 4.6(g) enumerates $\mathrm{O}\left(\sqrt[3]{R}\right)$ candidates for ℓ_1, instead of $\mathrm{O}(R)$ as in the worst-case analysis. Similarly, lines 4.6(h–i) enumerate the $\mathrm{O}\left(\sqrt[3]{R} \times \sqrt[3]{R}\right) = \mathrm{O}\left(R^{\frac{2}{3}}\right)$ candidates for each of h and t. Thus there are a total of

$$\mathrm{O}\left(\sqrt[3]{R}\right) \times \mathrm{O}\left(R^{\frac{2}{3}}\right) \times \mathrm{O}\left(R^{\frac{2}{3}}\right) = \mathrm{O}\left(R^{\frac{5}{3}}\right) \tag{4.4}$$

combinations of candidates for which **C3** must be evaluated. As before, evaluating **C3** involves $\mathrm{O}\left(M_{\max}N\right)$ invocations of the string-search operator. But each such invocation now involves pairs of strings with lengths bounded by $\sqrt[3]{R}$, and thus the total time to evaluate **C3** once is $\mathrm{O}\left(M_{\max}N\sqrt[3]{R}\right)$. Combining these results, we obtain the following heuristic-case bound for the running time lines 4.6(g–j)

$$\mathrm{O}\left(R^{\frac{5}{3}}\right) \times \mathrm{O}\left(M_{\max}N\sqrt[3]{R}\right) = \mathrm{O}\left(M_{\max}NR^2\right).$$

As in the worst-case analysis, the rest of $\mathsf{Generalize}_{\mathrm{HLRT}}$ runs in time bounded by lower-order polynomials of R and so these terms can be largely ignored. However, the rest of the algorithm introduces a linear dependency on K. Thus we have proved the following theorem.

Theorem 4.7 ($\mathsf{Generalize}^{*}_{\mathrm{HLRT}}$ heuristic-case complexity) *Under Assumption 4.1, the $\mathsf{Generalize}^{*}_{\mathrm{HLRT}}$ algorithm runs in time $O\left(KM_{\max}NR^2\right)$.*

Of course, the validity of this bound rests on Assumption 4.1. In Section 7.5, we demonstrate empirically that Assumption 4.1 does in fact hold in practice. We examined several actual Internet information resources, and found that the best-fit predictor of the partition element lengths is $R^{0.32} = \sqrt[3.1]{R}$. Thus Assumption 4.1 is validated by empirical evidence.

4.6 PAC analysis

In this section we apply the PAC model introduced in Section 3.2.3 to the problem of learning HLRT wrappers.

Following the PAC model, we assume that each example page P is drawn from a fixed but arbitrary and unknown probability distribution $\mathcal{D}$. Wrapper induction involves finding an approximation to the target HLRT wrapper $\mathcal{T}$. The Induce learning algorithm sees only $\mathcal{T}$'s behavior on the examples: the learner gathers a set $\mathcal{E} = \{\ldots, \langle P, \mathcal{T}(P)\rangle, \ldots\}$ of examples. Induce might not be able to isolate the target exactly, and therefore we require only that, with high probability, the learning algorithm find a good approximation to $\mathcal{T}$. The quality of an approximation to $\mathcal{T}$ is measured in terms of its error: $E_{\mathcal{T},\mathcal{D}}(w)$ is the chance (with respect to $\mathcal{D}$) that wrapper w makes a mistake (with respect to $\mathcal{T}$) on a single instance:

$$E_{\mathcal{T},\mathcal{D}}(w) = \mathcal{D}\left[P | w(P) \neq \mathcal{T}(P)\right]$$

The user supplies two numeric parameters: an accuracy parameter $0 < \epsilon < 1$ and a reliability parameter $0 < \delta < 1$. The basic question we want to answer is: How many examples does the $\mathsf{Generalize}_{\mathrm{HLRT}}$ induction algorithm need to see so that, with probability at least $1 - \delta$, it outputs a wrapper w that obeys $E_{\mathcal{T},\mathcal{D}}(w) < \epsilon$?

Formal result. This question is answered by the following theorem.

> **Theorem 4.8 (HLRT is PAC-learnable)** *Provided that Assumption 4.1 holds, the following property holds for any* $0 < \epsilon$, $\delta < 1$, *target wrapper* $\mathcal{T} \in \mathcal{H}_{\mathrm{HLRT}}$, *and page distribution* $\mathcal{D}$. *Suppose* $\mathsf{Generalize}_{\mathrm{HLRT}}$ *is given as input a set* $\mathcal{E} = \{\langle P_1, L_1\rangle, \ldots, \langle P_N, L_N\rangle\}$ *of example pages, each drawn*

independently according to distribution D and then labeled according to target T. If

$$\Psi(K)\left(1-\frac{\epsilon}{\Psi(K)}\right)^{M_{\mathrm{tot}}} \;+\; \Phi(R)\left(1-\frac{\epsilon}{2}\right)^{N} \quad < \quad \delta, \tag{4.5}$$

then $E_{T,D}(w) < \epsilon$ with probability at least $1-\delta$. The parameters are as follows: $\mathcal{E}$ consists of $N = |\mathcal{E}|$ examples, $M_{\mathrm{tot}} = \sum_n M_n$ is the total number of tuples in $\mathcal{E}$, page P_n contains $M_n = |L_n|$ tuples, each tuple consists of K attributes, the shortest example page has length $R = \min_n |P_n|$,

$$\Psi(K) = 4K - 2, \tag{4.6}$$

and

$$\Phi(R) = \frac{1}{4}\left(R^{\frac{5}{3}} - 2R^{\frac{4}{3}} + R\right). \tag{4.7}$$

The proof of Theorem 4.8 appears in Appendix B.4. In the remainder of this section, we interpret and evaluate this result.

Interpretation. We begin by applying Theorem 4.8 to an example. Consider an information resource presenting $K = 4$ attributes per tuple, an average of five tuples per example page (*i.e.*, $\frac{M_{\mathrm{tot}}}{N} = 5$), and where the shortest page is $R = 5000$ characters long. For the parameters $\epsilon = \delta = 0.1$, Theorem 4.8 states that $\mathsf{Generalize}_{\mathrm{HLRT}}$ must examine at least $N \geq 295$ example pages to satisfy the PAC criteria, while for $\epsilon = \delta = 0.01$, the bound grows to $N \geq 3476$.

Equation 4.5 can be understood as follows. The left-hand side is an upper bound on the chance that the examples will yield a wrapper with excessive error, while the right-hand side is the desired reliability level. Thus, requiring that the right-hand side be less than the left-hand side ensures that the PAC termination criterion is satisfied.

In standard PAC analyses, such an inequality is usually solved for the variable over which the learner has control, the number of examples $N = |\mathcal{E}|$. In Equation 4.5, there are three such variables: N, M_{tot}, and R. To understand why our result has this form, recall the fundamental point of PAC analysis. The goal is to examine the examples $\mathcal{E}$ and determine—on the basis of just N—whether the PAC termination criteria is satisfied. As the proof of Theorem 4.8 shows, it turns out that we need to examine $\mathcal{E}$ in more detail. That is, we simply can't be sure that the PAC termination criteria will be satisfied merely on the basis of N. Rather, we must also take into account the total number of tuples (M_{tot}) and the length of shortest example (R).

In more detail, the first term of the left-hand side of Equation 4.5,

$$\Psi(K)\left(1-\frac{\epsilon}{\Psi(K)}\right)^{M_{\mathrm{tot}}}, \tag{4.8}$$

is a bound on the chance that the $2K-1$ left and right delimiters (the ℓ_k and r_k, except for l_1) have been learned incorrectly. The second term,

$$\Phi(R)\left(1-\frac{\epsilon}{2}\right)^{N}, \tag{4.9}$$

is a bound on the chance that the remaining three delimiters (h, t, and ℓ_1) have been learned incorrectly. The overall chance that the wrapper is wrong is just the chance of the disjunction of these two events, and so the chance of learning the wrapper incorrectly is at most the sum of these two terms. The proof of Theorem 4.8 mainly involves deriving these two terms.

How the model behaves. To help visualize Equation 4.5, Figure 4.7 shows two projections of the multi-dimensional surface it describes. We have plotted the chance of successful learning as a function of the number of examples. More precisely, the graphs display the confidence that the learned wrapper has error at most ϵ—computed as one minus the left-hand side of Equation 4.5[5]—as a function of two variables: N,

[5] Actually, recall that the left-hand side of Equation 4.5 (call this quantity $c(N, M_{\mathrm{tot}})$) is only an *approximation* to the true confidence. As N and M_{tot} approach zero, the approximation degrades, so that eventually the "probability" $c(N, M_{\mathrm{tot}})$ exceeds one. Therefore, rather than plot $1-c(N, M_{\mathrm{tot}})$, we have plotted $1-\min(1, c(N, M_{\mathrm{tot}}))$.

the number of example pages, and $M_{\text{ave}} = \frac{M_{\text{tot}}}{N}$, the average number of tuples per example. (The surface is plotted using M_{ave} rather than M_{tot} because in practice, the average number of tuples per page is more meaningful than the total.) The remaining domain characteristics are set as in the earlier example: K is held constant at four attributes per tuple, and R is held constant at 5000 characters.

Figure 4.7(a) illustrates the surface for $\epsilon = 0.1$, and (b) illustrates the surface for $\epsilon = 0.01$. In each case, as N and M_{ave} increase, the PAC confidence asymptotically approaches one. We have also indicated the confidence level 0.9; points above this threshold correspond to learning trials in which the PAC termination criterion is satisfied for the reliability parameter $\delta = 0.1$ Earlier, we reported that a learner must examine at least $N \geq 295$ examples for $\epsilon = \delta = 0.1$ and $N \geq 3476$ examples for $\epsilon = \delta = 0.01$; these graphs show that these values are correct.

Evaluation. The PAC model is just that—a *model*—and thus an important issue is whether Theorem 4.8 provides an *effective* termination condition. Informally, we want to know whether the PAC model terminates the learning process too soon (meaning that more examples ought to be considered before stopping) or too late (fewer examples are sufficient for a high-quality wrapper).

More precisely, the PAC model would terminate the learning algorithm too soon if its confidence were too high. This could happen because the proof of Theorem 4.8 relies on the assumption that the examples are drawn *independently* from the distribution $\mathcal{D}$. If this assumption does not hold, then the model is overconfident, because the observed examples are not in fact representative of $\mathcal{D}$.

On the other hand, the PAC model would terminate the learning algorithm too late if its confidence were too low. This could happen because Theorem 4.8 makes *no* assumptions about the instance distribution $\mathcal{D}$. Indeed, the PAC model is essentially a worst-case analysis of $\mathcal{D}$. It is assumed, for example, that the chance of learning r_1 in no way effects the chance of learning r_2. But it might be the case that two

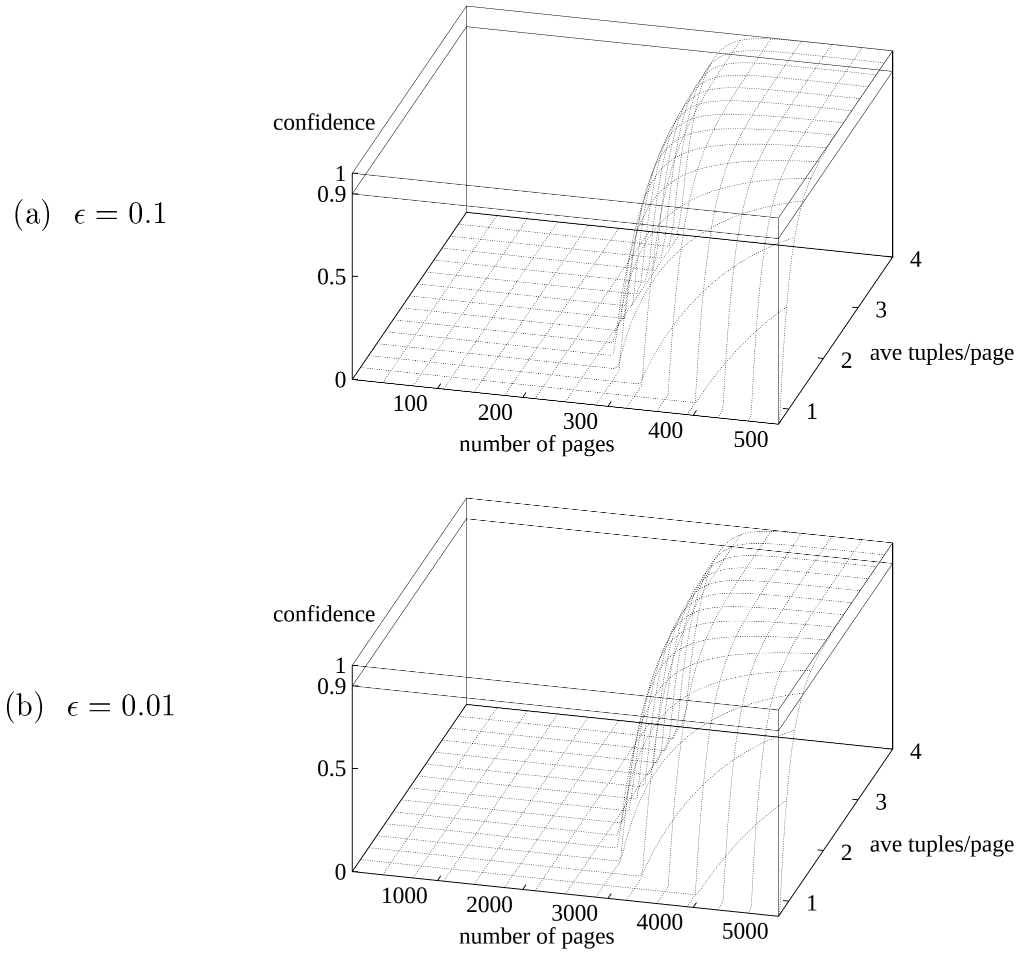


Figure 4.7: *Surfaces showing the confidence that a learned wrapper has error at most ϵ, as a function of N (the total number of examples pages) and $M_{\text{ave}} = \frac{M_{\text{tot}}}{N}$ (the average number of tuples per example), for* (a) $\epsilon = 0.1$ *and* (b) $\epsilon = 0.01$.

events are in fact probabilistically dependent under $\mathcal{D}$. For example, there may be a set of "pedagogically useful" instances such that the learner will succeed if it sees any one. In summary, it may be the case that the distributions actually encountered in practice are substantially easier to learn from than the worst case.

Though analytic techniques could in principle be applied to determine how well the PAC model fits the actual learning task, we have taken an empirical approach to validating the model. Section 7.4 demonstrates that for several real HLRT learning tasks, the PAC model stops the induction process substantially later than is actually necessary, even with relatively weak reliability and accuracy parameters levels of $\epsilon = \delta = 0.1$. Tightening this prediction is a challenging direction for future research.

A simpler PAC model. As described earlier, the left-hand side of the Equation 4.5 is the sum of two terms, listed above as Equations 4.8 and 4.9:

$$\Psi(K)\left(1 - \frac{\epsilon}{\Psi(K)}\right)^{M_{\text{tot}}} \;+\; \Phi(R)\left(1 - \frac{\epsilon}{2}\right)^{N}.$$

We now show that this sum tends to be dominated by the second term, so that the PAC model reduces to a much simpler form, which in turn leads to an interesting theoretical result.

Note that typically $\Psi(K) \ll \Phi(R)$, since $K \ll R$; for instance if $K = 4$ and $R = 5000$, then $\Psi(K) = 14$ while $\Psi(R) > 10^5$. Moreover, the difference between $1 - \frac{\epsilon}{2}$ and $1 - \frac{\epsilon}{\Phi(K)}$ is usually small; for example, with $K = 4$ and $\epsilon = 0.1$, these terms differ by about 4%. Finally, typically each example page has many tuples and therefore $M_{\text{tot}} \gg N$, so that for any $0 < \eta < 1$, $\eta^{M_{\text{tot}}} \ll \eta^N$.

Putting these observations together, we can reason informally as follows:

$$\begin{aligned}
&\Psi(K)\left(1 - \frac{\epsilon}{\Psi(K)}\right)^{M_{\text{tot}}} \;+\; \Phi(R)\left(1 - \frac{\epsilon}{2}\right)^{N} \\
\approx\;\; & \text{small}\left(1 - \frac{\epsilon}{\Psi(K)}\right)^{M_{\text{tot}}} \;+\; \Phi(R)\left(1 - \frac{\epsilon}{2}\right)^{N} \\
\approx\;\; & \text{small}\left(1 - \frac{\epsilon}{2}\right)^{M_{\text{tot}}} \;+\; \Phi(R)\left(1 - \frac{\epsilon}{2}\right)^{N} \\
\approx\;\; & \text{small}\cdot\text{small} \;+\; \Phi(R)\left(1 - \frac{\epsilon}{2}\right)^{N} \\
\approx\;\; & \Phi(R)\left(1 - \frac{\epsilon}{2}\right)^{N}.
\end{aligned}$$

To summarize, if $\Psi(K) \ll \Phi(R)$ and $N \ll M_{\text{tot}}$, then our PAC model (Equation 4.5) simplifies to

$$\Phi(R)\left(1-\frac{\epsilon}{2}\right)^{N} \quad < \quad \delta. \tag{4.10}$$

For example, using $K = 4$, $R = 5000$, $\epsilon = 0.1$, $M_{\text{ave}} = 5$, and $N = 200$, the original PAC confidence and this approximation differ by less 0.5%.

This simplified model yields an interesting theoretical result, and thus we state the assumptions behind it explicitly.

Assumption 4.2 (Few attributes, plentiful data) *For any set of examples $\mathcal{E} = \{\ldots, \langle P_n, L_n\rangle, \ldots\}$, we have that*

$$\Psi(K)\left(1-\frac{\epsilon}{\Psi(K)}\right)^{M_{\text{tot}}} \quad \ll \quad \Phi(R)\left(1-\frac{\epsilon}{2}\right)^{N}$$

where $N = |\mathcal{E}|$, each tuple has K attributes, $R = \min_n |P_n|$ is the length of the shortest page, the examples together have M_{tot} tuples, and $\Psi(K)$ and $\Phi(R)$ are as defined in Theorem 4.8.

We can now solve Equation 4.10 for N. Doing so gives us a bound on the number of examples needed to ensure that a the learned wrapper is PAC. Using the inequality $(1-x) \leq e^{-x}$, we have that

$$N > \frac{2}{\epsilon}\log\frac{\Phi(R)}{\delta}. \tag{4.11}$$

Since (1) N is polynomial in $\frac{1}{\epsilon}$ and $\frac{1}{\delta}$ in Equation 4.11, and (2) $\mathsf{Generalize}_{\text{HLRT}}$ runs in quadratic time (Theorem 4.7) we have therefore established the following result.[6]

Theorem 4.9 *Wrapper class $\mathcal{H}_{\text{HLRT}}$ is efficiently PAC-learnable under Assumption 4.2.*

This simpler PAC model makes intuitive sense. The bottleneck when learning HLRT wrappers is an adequate supply of page heads and tails; each example provides just one of each. In contrast, the body of a page typically contains several tuples and

[6] Recall Footnote 2 on page 33. Here R plays the role of the natural complexity measure of the HLRT hypothesis class; notice that N is polynomial in R as well.

thus several opportunities for learning the r_k and ℓ_k delimiters. Therefore, if we can count on each page to provide several tuples, then collectively the examples provide many more tuples than page heads or tails. Thus the part of the PAC model related to learning from the page bodies (Equation 4.8) is insignificant compared to the part of the model related to learning from the page heads and tails (Equation 4.9).

Of course, Theorem 4.9 holds only if Assumption 4.2 is realistic. In Section 7.6, we take an empirical approach to verifying Assumption 4.2. We demonstrate that, for several actual experiments, the deviation between the simplified and original PAC models is small.

Finally, notice that the variable K does not occur in Equation 4.11. Therefore, under Assumption 4.2, the number of examples required to satisfy the PAC criteria is independent of K. This is an interesting theoretical result, since it suggests that the techniques developed in this thesis can be applied to information resources that contain arbitrarily many attributes.

4.7 Summary

In this chapter we have seen our first complete instance of how to learn a wrapper class. Conceptually, this was quite straightforward: we defined the HLRT wrapper class; developed $\mathsf{Generalize}_{\mathrm{HLRT}}$, the generalization function input to the Induce algorithm; and designed a PAC-theoretic model of the HLRT wrapper class.

Along the way, we examined many technical details. Our straightforward implementation of $\mathsf{Generalize}_{\mathrm{HLRT}}$ is extremely slow, so we developed a substantially more efficient variant, $\mathsf{Generalize}^*_{\mathrm{HLRT}}$. We were able to reason about these algorithms because we explicitly wrote down the constraint $\mathcal{C}_{\mathrm{HLRT}}$ governing when a wrapper is consistent with an example. We then went on to perform an heuristic-case analysis of our learning algorithm, which reveals that our algorithm runs in time quadratic in the relevant parameters.

We then developed a PAC model. Since HLRT wrappers have quite a complex structure, the PAC bound is necessarily rather complex as well. Interestingly, our model reduces to a very simple form under conditions that are easily satisfied by actual Internet resources. Specifically, we have found that, under reasonable assumptions, the number of examples required to learn an HLRT wrapper does not depend on the number of attributes (K), and that our induction algorithm requires only a polynomial-sized collection of examples in order to learn effectively.

HLRT is just one of many possible wrapper classes. For example, we have already mentioned a simplification of HLRT, which we call LR. In Chapter 5, we discuss the automatic induction of LR and four other wrapper classes.

Chapter 5

BEYOND HLRT: ALTERNATIVE WRAPPER CLASSES

5.1 Introduction

So far, we have been concerned exclusively with the HLRT wrapper class. This suggests a natural question: what about other wrapper classes? In the extreme, wrappers can be unrestricted programs in a fully general programming language. Of course automatically learning such wrappers would be difficult.

In this chapter we describe five additional wrapper classes for which we have found efficient learning algorithms. Like HLRT, all are based on the idea of using delimiters that mark the left- and right-hand side of the text fragments to be extracted. The classes differ from HLRT in two ways. First, we developed various techniques to avoid getting confused by distractions such as advertisements (Section 5.2). Second, we developed wrapper classes for extracting information that is laid out not as a table (the structure assumed by HLRT), but rather as a hierarchically nested structure (*e.g.*, a book's table of contents) (Section 5.3).

5.2 Tabular resources

We first consider alternatives to HLRT. As discussed in Chapter 4, the HLRT wrapper class corresponds to one particular "programming idiom" for writing wrappers. In this section we consider alternatives.

5.2.1 *The* LR, OCLR *and* HOCLRT *wrapper classes*

The HLRT wrapper class. We begin by quickly reviewing the HLRT wrapper class. HLRT wrappers use one component (h) to skip over a page's head, and a second (t) indicates a page's tail. In the page's body, a pair of left- and right-hand delimiters (ℓ_k and r_k) is used to extract each of the K attribute values for each tuple.

We have seen that we can encapsulate the behavior of an HLRT wrapper for a domain with K attributes as a vector $\langle h, t, \ell_1, r_1, \ldots, \ell_K, r_K\rangle$ of $2K+2$ strings. The meaning of these strings is defined by the ExecHLRT procedure, which executes an HLRT wrapper on a given page; see Figure 4.1.

The LR wrapper class. The LR wrapper class is a simplification of HLRT. In a nutshell, LR is simply HLRT without the "H" or "T". In pseudo-code, LR execution proceeds as follows:

```
ExecLR(wrapper $\langle \ell_1, r_1, \ldots, \ell_K, r_K\rangle$, page $P$)
    while there is a next occurrence of $\ell_1$ in $P$
        for each $\langle \ell_k, r_k\rangle \in \{\langle \ell_1, r_1\rangle, \ldots, \langle \ell_K, r_K\rangle\}$
            extract from $P$ the value of the next instance of the $k^{\text{th}}$ attribute
            between the next occurrence of $\ell_k$ and the subsequent occurrence of $r_k$
    return all extracted tuples
```

More precisely, we define the execution of an LR wrapper as follows:

```
ExecLR(wrapper $\langle \ell_1, \ell_K, \ldots, \ell_K, r_K\rangle$, page $P$)
    $i \leftarrow 0$
    $m \leftarrow 0$
    while $P[i]/\ell_1 \neq \Diamond$                              (a)
        $m \leftarrow m+1$
        for each $\langle \ell_k, r_k\rangle \in \{\langle \ell_1, r_1\rangle, \ldots, \langle \ell_K, r_K\rangle\}$
            $i \leftarrow i + P[i]\#\ell_k + |\ell_k|$                   (b)
            $b_{m,k} \leftarrow i$
            $i \leftarrow i + P[i]\#r_k$
            $e_{m,k} \leftarrow i - 1$
    return label $\{\ldots, \langle \ldots, \langle b_{m,k}, e_{m,k}\rangle, \ldots\rangle, \ldots\}$
```

Just as an HLRT wrapper can be described exactly as a vector of $2K+2$ strings, the behavior of an LR wrapper can be encapsulated entirely as a vector of $2K$ strings

$\langle \ell_1, r_1, \ldots, \ell_K, r_K \rangle$.

For example, consider the following page (artificially produced for the sake of illustration):

$$\texttt{ho[A11](A12)co[A21](A22)co[A31](A32)ct}$$

The task is to label this page so as to extract the three tuples of two attributes,

$$\{\langle \texttt{A11}, \texttt{A12} \rangle, \langle \texttt{A21}, \texttt{A22} \rangle, \langle \texttt{A31}, \texttt{A32} \rangle\}$$

The HLRT wrapper $\langle \texttt{h}, \texttt{t}, \texttt{[}, \texttt{]}, \texttt{(}, \texttt{)} \rangle$ as well as the LR wrapper $\langle \texttt{[}, \texttt{]}, \texttt{(}, \texttt{)} \rangle$ both are consistent with—*i.e.*, extract the given information from—this example page.[1]

The OCLR wrapper class. HLRT employs the delimiters h and t to prevent incorrect extraction from a page's head and tail. The OCLR wrapper class provides a different mechanism for handling a similar problem.

Rather than treating a page as "head plus body plus tail", the OCLR wrapper treats the page as a sequence of tuples separated by irrelevant text. OCLR is designed to avoid distracting irrelevant text in these inter-tuple regions (just as HLRT is designed to avoid distracting text in the head and tail).

OCLR uses two strings, one (denoted o) that marks the beginning or *opening* of each tuple, and a second (denoted c) that marks then end or *closing* of each tuple. OCLR uses o and c to indicate the opening and closing of each tuple, much as HLRT and LR use ℓ_k and r_k mark the beginning and end of the $k^{\underline{\text{th}}}$ attribute.

As with LR and HLRT, we can describe the behavior of an OCLR wrapper using a vector $\langle o, c, \ell_1, r_1, \ldots, \ell_K, r_K \rangle$ of $2K + 2$ strings. To make the meaning of these strings precise, we must describe the ExecOCLR procedure. ExecOCLR prescribes the meaning of an OCLR wrapper $\langle o, c, \ell_1, r_1, \ldots, \ell_K, r_K \rangle$, just as ExecHLRT defines HLRT wrappers, and ExecLR defines LR wrappers. In pseudo-code:

[1] Note that we use a `fixed-width` font to indicate the actual fragments of a page such as "`h`", while *italics* indicates HLRT wrapper components such as "h".

```
ExecOCLR(wrapper $\langle o, c, \ell_1, r_1, \ldots, \ell_K, r_K \rangle$, page $P$)
    while there is a next occurrence of $o$ in $P$
        skip to the next occurrence of $o$ in $P$
        for each $\langle \ell_k, r_k \rangle \in \{\langle \ell_1, r_1 \rangle, \ldots, \langle \ell_K, r_K \rangle\}$
            extract from $P$ the value of the next instance of the $k^{\underline{\text{th}}}$ attribute
            between the next occurrence of $\ell_k$ and the subsequent occurrence of $r_k$
        skip past the next occurrence of $c$ in $P$
    return all extracted tuples
```

More precisely, we define the execution of an OCLR wrapper as follows:

```
ExecOCLR(wrapper $\langle o, c, \ell_1, r_1, \ldots, \ell_K, r_K \rangle$, page $P$)
    $i \leftarrow 0$
    $m \leftarrow 0$
    while $P[i]/o \neq \Diamond$                                        (a)
        $m \leftarrow m + 1$
        $i \leftarrow i + P[i]\#o$
        for each $\langle \ell_k, r_k \rangle \in \{\langle \ell_1, r_1 \rangle, \ldots, \langle \ell_K, r_K \rangle\}$
            $i \leftarrow i + P[i]\#\ell_k + |\ell_k|$                  (b)
            $b_{m,k} \leftarrow i$
            $i \leftarrow i + P[i]\#r_k$
            $e_{m,k} \leftarrow i - 1$
        $i \leftarrow i + P[i]\#c$                                      (c)
    return label $\{\ldots, \langle \ldots, \langle b_{m,k}, e_{m,k} \rangle, \ldots \rangle, \ldots\}$
```

To illustrate OCLR, suppose that the following HTML page was produced by a resource similar to the original country/code example (Figure 2.1):

```
<HTML><TITLE>Some Country Codes</TITLE><BODY>
<B>1</B> <B>Congo</B> <I>242</I><BR>
<B>2</B> <B>Egypt</B> <I>20</I><BR>
<B>3</B> <B>Belize</B> <I>501</I><BR>
<B>4</B> <B>Spain</B> <I>34</I><BR>
</BODY></HTML>
```

In the original example, every LR wrapper will get confused, because distracting text in the page's head and tail is incorrectly extracted as a country. Here, the problem is that text *between* the tuples is marked up using the same tags as are used for countries. One way to avoid this problem is to use an OCLR wrapper with $o =$ `</B>` and $c =$ `<BR>`. When the OCLR wrapper $\langle$`</B>`, `<BR>`, `<B>`, `</B>`, `<I>`, `</I>`$\rangle$ is given to ExecOCLR, these c and o tags force the irrelevant parts of the page to be skipped.[2]

[2] It is true that there exists an LR wrapper that can handle this modification to the country/code

As a second illustration of an OCLR wrapper, consider again the example page `ho[A11](A12)co[A21](A22)co[A31](A32)ct`. Along with the LR and HLRT wrappers mentioned earlier, the OCLR wrapper ⟨`o`, `c`, `[`, `]`, `(`, `)`⟩ is consistent with this page.

The HOCLRT wrapper class. As the ExecHOCLRT procedure pseudo-code illustrates, the HOCLRT wrapper class combines the functionality of OCLR and HLRT:

```
ExecHOCLRT(wrapper $\langle h, t, o, c, \ell_1, r_1, \ldots, \ell_K, r_K \rangle$, page $P$)
    skip past the first occurrence of $h$ in $P$
    while the next occurrence of $o$ is before the next occurrence of $t$ in $P$
        skip to the next occurrence of $o$ in $P$
        for each $\langle \ell_k, r_k \rangle \in \{\langle \ell_1, r_1 \rangle, \ldots, \langle \ell_K, r_K \rangle\}$
            extract from $P$ the value of the next instance of the $k^{\underline{\text{th}}}$ attribute
            between the next occurrence of $\ell_k$ and the subsequent occurrence of $r_k$
        skip past the next occurrence of $c$ in $P$
    return all extracted tuples
```

More precisely:

```
ExecHOCLRT(wrapper $\langle h, t, o, c, \ell_1, r_1, \ldots, \ell_K, r_K \rangle$, page $P$)
    $i \leftarrow P\#h$
    $m \leftarrow 0$
    while $|P[i]/t| > |P[i]/o|$                                    (a)
        $m \leftarrow m + 1$
        $i \leftarrow i + P[i]\#o$
        for each $\langle \ell_k, r_k \rangle \in \{\langle \ell_1, r_1 \rangle, \ldots, \langle \ell_K, r_K \rangle\}$
            $i \leftarrow i + P[i]\#\ell_k + |\ell_k|$                  (b)
            $b_{m,k} \leftarrow i$
            $i \leftarrow i + P[i]\#r_k$
            $e_{m,k} \leftarrow i - 1$
        $i \leftarrow i + P[i]\#c$                                    (c)
    return label $\{\ldots, \langle \ldots, \langle b_{m,k}, e_{m,k} \rangle, \ldots \rangle, \ldots\}$
```

We can use the example page `ho[A11](A12)co[A21](A22)co[A31](A32)ct` mentioned earlier to illustrate HOCLRT: wrapper ⟨`h`, `t`, `o`, `c`, `[`, `]`, `(`, `)`⟩ is consistent with this page.

example; namely, ⟨`</B> <B>`, `</B>`, `<I>`, `</I>`⟩. However, as we'll see in Section 5.2.3, there are pages that can be wrapped by OCLR but not LR. The purpose of this example is simply to illustrate OCLR, not to establish its superiority to LR.

5.2.2 Segue

Having introduced four wrapper classes, a natural question arises: on what basis can they be compared? In the next two sections, we provide two such bases: *relative expressiveness*, which measures the extent to which the funtionality of one wrapper class can be mimicked by another, and *learning complexity*, which measures the computational complexity of learning wrappers within each class.

5.2.3 Relative expressiveness

A key issue when comparing LR, HLRT, OCLR and HOCLRT concerns their relative expressiveness: which information resources can be handled by certain wrapper classes but not by others? For example, we have observed that the page

```
ho[A11](A12)co[A21](A22)co[A31](A32)ct
```

can be handled by all four wrapper classes. On the other hand, the country/code example (Figure 2.1) can be wrapped by HLRT but not by LR.

To formalize this investigation, let $\Pi = \{\ldots, \langle P, L\rangle, \ldots\}$ be the *resource space*, the set of all page/label pairs $\langle P, L\rangle$. Conceptually, Π includes pages from all information resources, whether regularly structured or unstructured, tabular or not, and so forth. For each such information resource, Π contains all of the resource's pages, and each page P included in Π is paired with its label L. Note that a particular information resource is equivalent to a subset of Π; namely, the particular page/label pairs the resource contains.

More importantly from the perspective of relative expressiveness, note that a wrapper class can be identified with a subset of Π: a class corresponds to those page/label pairs for which a consistent wrapper exists in the class. If $\mathcal{W}$ is a wrapper class, then we use the notation $\Pi(\mathcal{W})$ to indicate the subset of Π which can be handled by $\mathcal{W}$.

Definition 5.1 (Π, $\Pi(\mathcal{W})$) *The* resource space Π *is defined to be the set of all $\langle P, L\rangle$ page/label pairs.*

The subset of Π *that is* wrappable *by a particular wrapper class $\mathcal{W}$, written $\Pi(\mathcal{W})$, is defined as the set of pairs $\langle P, L\rangle \in \Pi$ such that a wrapper consistent with $\langle P, L\rangle$ exists in class $\mathcal{W}$:*

$$\Pi(\mathcal{W}) = \{\langle P, L\rangle \in \Pi \mid \exists_{w \in \mathcal{H}_{\mathcal{W}}} w(P) = L\}.$$

Note that $\Pi(\mathcal{W})$ provides a natural way to compare the relative expressiveness of wrapper classes. For example, let $\mathcal{W}_1$ and $\mathcal{W}_2$ be two wrapper classes. If $\Pi(\mathcal{W}_1) \subset \Pi(\mathcal{W}_2)$, then $\mathcal{W}_2$ is more expressive than $\mathcal{W}_1$, in the sense that any page that can be wrapped by $\mathcal{W}_1$ can also be wrapped by $\mathcal{W}_2$.

Figure 5.1 and Theorem 5.1 capture our results concerning the relative expressiveness of the four wrapper classes discussed in this section.

Theorem 5.1 (Relative expressiveness of LR, HLRT, OCLR and HOCLRT) *The relationships between* Π(LR), Π(HLRT), Π(OCLR) *and* Π(HOCLRT) *are as indicated in Figure 5.1.*

Proof of Theorem 5.1 (Sketch): To establish these relationships, it suffices to show that:

1. There exists at least one pair $\langle P, L\rangle \in \Pi$ in each of the regions marked *(A)*, *(B)*, *(C)*, …,*(I)* in Figure 5.1;
2. OCLR subsumes LR: Π(LR) $\subset$ Π(OCLR); and
3. HOCLRT subsumes HLRT: Π(HLRT) $\subset$ Π(HOCLRT).

Note that these three assertions jointly imply that the four wrapper classes are related as claimed.

Consider each assertion in turn.

1. In Appendix B.5 we identify one $\langle P, L\rangle$ pair in each of the regions *(A)*, *(B)*, *etc.*

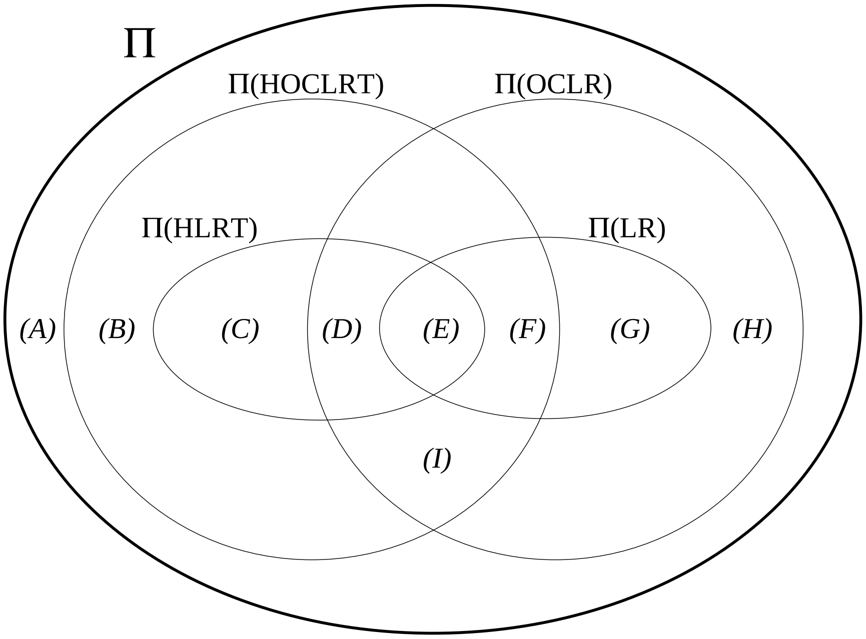


Figure 5.1: *The relative expressiveness of the* LR, HLRT, OCLR, *and* HOCLRT *wrapper classes.*

2. The idea is that an OCLR wrapper can always be constructed from an LR wrapper for any particular page. Specifically, suppose there exists a pair $\langle P, L\rangle \in \Pi(\textsc{lr})$—*i.e.*, there exists a wrapper $w = \langle \ell_1, r_1, \ldots, \ell_K, r_K\rangle \in \mathcal{H}_{\mathrm{LR}}$ such that $w(P) = L$. Then OCLR wrapper $w' = \langle \ell_1, \phi, \ell_1, r_1, \ldots, \ell_K, r_K\rangle$ satisfies $w'(P) = L$, and therefore $\langle P, L\rangle \in \Pi(\textsc{oclr})$.[3]
3. The same idea applies to the classes HLRT and HOCLRT. Suppose there exists a pair $\langle P, L\rangle \in \Pi(\textsc{hlrt})$—*i.e.*, there exists a wrapper $w = \langle h, t, \ell_1, r_1, \ldots, \ell_K, r_K\rangle \in \mathcal{H}_{\mathrm{HLRT}}$ such that $w(P) = L$. Then HOCLRT wrapper $w' = \langle h, t, \ell_1, \phi, \ell_1, r_1, \ldots, \ell_K, r_K\rangle$ satisfies $w'(P) = L$, and therefore $\langle P, L\rangle \in \Pi(\textsc{hoclrt})$.

See Appendix B.5 for complete details.

□ (Proof of Theorem 5.1 (Sketch))

[3] As described in Appendix C, the symbol 'ϕ' denotes the empty string.

One possibly counterintuitive implication of Theorem 5.1 is that the LR class is not subsumed by the HLRT class. One might expect that an HLRT wrapper can always be constructed to mimic the behavior of any given LR wrapper. To do so, the head delimiter can simply be set to the empty string: $h = \phi$. However, the tail delimiter t must be set some non-empty page fragment. In general such a delimiter might not exist. For similar reasons, the OCLR wrapper class is not subsumed by the HOCLRT class.

5.2.4 Complexity of learning

The previous section illustrated that the four wrapper classes under consideration cover different parts of the resource space Π. For example, LR covers strictly less of Π than OCLR. Thus a natural basis for comparing wrapper classes is the computational tradeoffs of these coverage differences. Since this thesis concerns learning wrappers, we are interested in the computational complexity of *learning* wrappers from the wrapper classes.

We will concentrate on the following scenario. Suppose we are given a set $\mathcal{E} = \{\ldots, \langle P_n, L_n \rangle, \ldots\}$ of examples. Let $N = |\mathcal{E}|$ be the number of examples. Suppose each tuple consists of K attributes, and let $M_{\text{max}} = \max_n M_n$ be the maximum number of tuples on any single page, where $M_n = |L_n|$ is the number of tuples on page P_n. Finally, suppose that the shortest example has length $R = \min_n |P_n|$. For each wrapper class $\mathcal{W}$, recall that $\mathsf{Generalize}_{\mathcal{W}}$ function is the input to the Induce learning algorithm when learning wrapper class $\mathcal{W}$. To measure the complexity of learning a particular class $\mathcal{W}$, we will be interested in the heuristic-case (defined by Assumption 4.1) running time of the function call $\mathsf{Generalize}_{\mathcal{W}}(\mathcal{E})$.

The HLRT wrapper class. We have already established (Theorem 4.7) that, under Assumption 4.1, the $\mathsf{Generalize}_{\text{HLRT}}$ function runs in time $\mathrm{O}\left(K M_{\text{max}} N R^2\right)$.

The LR wrapper class. The LR generalization function, $\mathsf{Generalize}_{\mathrm{LR}}$, is similar to $\mathsf{Generalize}_{\mathrm{HLRT}}$, except that a different consistency constraint—$\mathcal{C}_{\mathrm{LR}}$ instead of $\mathcal{C}_{\mathrm{HLRT}}$—must be satisfied.

The consistency constraints for the LR wrapper class is the predicate $\mathcal{C}_{\mathrm{LR}}$. Let $w = \langle \ell_1, r_1, \ldots, \ell_K, r_K \rangle$ be an LR wrapper, and $\langle P, L \rangle$ be a pair/label pair. We define $\mathcal{C}_{\mathrm{LR}}$ as follows:

$$\begin{aligned} \mathcal{C}_{\mathrm{LR}}(w, \langle P, L \rangle) \quad \Longleftrightarrow \quad & \bigwedge_{1 \le k \le K} \mathbf{C1}(r_k, \langle P, L \rangle) \\ \wedge \quad & \bigwedge_{1 \le k \le K} \mathbf{C2}(\ell_k, \langle P, L \rangle). \end{aligned}$$

The difference between HLRT and LR is mirrored in the difference between $\mathcal{C}_{\mathrm{HLRT}}$ and $\mathcal{C}_{\mathrm{LR}}$. While HLRT wrappers must satisfy constraint **C3** for the h, t, and ℓ_1 components, LR wrappers can entirely ignore this constraint. Instead, LR requires simply that, like the other $(K-1)$ ℓ_k components, ℓ_1 must satisfy **C2**. In particular, notice that the "$1 < k \le K$" quantifier in the **C2** conjunct of $\mathcal{C}_{\mathrm{HLRT}}$ is replaced by the quantifier "$1 \le k \le K$" in $\mathcal{C}_{\mathrm{LR}}$.

As with HLRT, we proceed by establishing that $\mathcal{C}_{\mathrm{LR}}$ is correct:

> **Theorem 5.2 ($\mathcal{C}_{\mathrm{LR}}$ is correct)** *For every* LR *wrapper w, page P, and label L, $\mathcal{C}_{\mathrm{LR}}(w, \langle P, L \rangle) \iff \mathsf{ExecLR}(w, P) = L$.*

The proof is similar (though simpler) than that of Theorem 4.1. The details are omitted.

Given $\mathcal{C}_{\mathrm{LR}}$, the generalization function $\mathsf{Generalize}_{\mathrm{LR}}$ is as follows:

```
$\underline{\mathsf{Generalize}_{\mathrm{LR}}}$(examples $\mathcal{E} = \{\langle P_1, L_1 \rangle, \ldots, \langle P_N, L_N \rangle\}$)
  for each $1 \le k \le K$
    for $r_k \leftarrow$ each prefix of $P_1$'s intra-tuple separator for $S_{1,k}$
      accept $r_k$ if $\mathbf{C1}$ holds of $r_k$ and every $\langle P_n, L_n \rangle \in \mathcal{E}$          (i)
  for each $1 \le k \le K$
    for $\ell_k \leftarrow$ each suffix of $P_1$'s intra-tuple separator $S_{1,k-1}$ ($S_{1,K}$ when $k = 1$)
      accept $\ell_k$ if $\mathbf{C2}$ holds of $\ell_k$ and every $\langle P_n, L_n \rangle \in \mathcal{E}$          (ii)
  return $\langle \ell_1, r_1, \ldots, \ell_K, r_K \rangle$
```

To see that $\mathsf{Generalize}_{\mathrm{LR}}$ is correct, recall that predicates **C1** and **C2** are independent, in that each r_k is constrained only by **C1**, while each ℓ_k depends only on **C2**. Thus $\mathsf{Generalize}_{\mathrm{LR}}$ is complete even though it does not consider all possible combinations for all values of the $2K$ r_k and ℓ_k components. Thus we have given a proof sketch for the following:

Theorem 5.3 $\mathsf{Generalize}_{\mathrm{LR}}$ *is consistent.*

(This result is analogous to Theorem 4.2 for the HLRT wrapper class.)

What is the complexity of $\mathsf{Generalize}_{\mathrm{LR}}$? The analysis is similar to that for Theorem 4.7. Lines (i) and (ii) of $\mathsf{Generalize}_{\mathrm{LR}}$ are each executed $\mathrm{O}\left(K\sqrt[3]{R}\right)$ times. Predicates **C1** and **C2** each require the same amount of time: $\mathrm{O}\left(M_{\max}N\sqrt[3]{R}\right)$. Thus the total time to execute $\mathsf{Generalize}_{\mathrm{LR}}$ is

$$2 \times \mathrm{O}\left(K\sqrt[3]{R}\right) \times \mathrm{O}\left(M_{\max}N\sqrt[3]{R}\right) = \mathrm{O}\left(KM_{\max}NR^{\frac{2}{3}}\right).$$

Thus we have established the following theorem:

Theorem 5.4 ($\mathsf{Generalize}_{\mathrm{LR}}$ heuristic-case complexity) *Under Assumption 4.1, the* $\mathsf{Generalize}_{\mathrm{LR}}$ *algorithm runs in time* $O\left(KM_{\max}NR^{\frac{2}{3}}\right)$.

The OCLR wrapper class. In the HLRT wrapper class, the three components h, t and ℓ_1 *interact*, in that they are jointly governed by the predicate **C3**. Similarly, in OCLR, the three components o, c, and ℓ_1 interact.

The OCLR consistency constraint predicate for wrapper $w = \langle o, c, \ell_1, r_1, \ldots, \ell_K, r_K\rangle$ and example $\langle P, L\rangle$ is defined as follows:

$$\begin{aligned}\mathcal{C}_{\mathrm{OCLR}}(w, \langle P, L\rangle) \quad \Longleftrightarrow \quad & \bigwedge_{1\le k\le K} \mathbf{C1}(r_k, \langle P, L\rangle) \\ \wedge \quad & \bigwedge_{1<k\le K} \mathbf{C2}(\ell_k, \langle P, L\rangle) \\ \wedge \quad & \mathbf{C4}(o, c, \ell_1, \langle P, L\rangle).\end{aligned}$$

Notice that, like HLRT but unlike LR, the **C2** conjunct of $\mathcal{C}_{\text{OCLR}}$ is quantified by "$1 < k \leq K$", since ℓ_1 is governed by **C4** rather than **C2**.

Predicate **C4** governs the interaction between o, c, and ℓ_1 for the OCLR wrapper class, just as **C3** governs h, t, and ℓ_1 for HLRT. For o, c, and ℓ_1 to satisfy **C4** for a particular example, it must be the case that: **(i)** in the page's head, o must skip over any potentially confusing material, so that ℓ_1 indicates the beginning of the first attribute of the first tuple; **(ii)** in the page's tail, c must mark the end of the last tuple and there must be no o present indicating a subsequent tuple; and **(iii)** between each tuple, c and o must together skip over any potentially confusing material so that ℓ_1 indicates the beginning of the first attribute of the next tuple. More formally:

$$\begin{array}{rcll} \mathbf{C4}(o, c, \ell_1, \langle P, L\rangle) & \iff & (S_{0,K}/o)/\ell_1 = \ell_1 & \textbf{(i)} \\ & & \wedge \;\; S_{M,K}/c \neq \Diamond \;\; \wedge \;\; (S_{M,K}/c)/o = \Diamond & \textbf{(ii)} \\ & & \wedge \;\; \forall_{1\leq m<M} \; ((S_{m,K}/c)/o)/\ell_1 = \ell_1. & \textbf{(iii)} \end{array}$$

As with HLRT and LR, before proceeding we must establish the following theorem:

Theorem 5.5 ($\mathcal{C}_{\text{OCLR}}$ is correct) *For every* OCLR *wrapper w, page P, and label L, $\mathcal{C}_{\text{OCLR}}(w, \langle P, L\rangle) \iff \mathsf{ExecOCLR}(w, P) = L$.*

The proof is similar to that of Theorems 4.1 and 5.2: we show that under no circumstances can ExecOCLR produce the wrong answer if $\mathcal{C}_{\text{OCLR}}$ is satisfied, and furthermore that if the wrapper is correct, then $\mathcal{C}_{\text{OCLR}}$ must hold. The details are omitted.

To analyze the complexity of learning OCLR, we must state its generalization function:

```
Generalize_OCLR(examples $\mathcal{E} = \{\langle P_1, L_1\rangle, \ldots, \langle P_N, L_N\rangle\}$)
    for each $1 \leq k \leq K$
      for $r_k \leftarrow$ each prefix of $P_1$'s intra-tuple separator for $S_{1,k}$
        accept $r_k$ if C1 holds of $r_k$ and every $\langle P_n, L_n\rangle \in \mathcal{E}$                (i)
    for each $1 < k \leq K$
      for $\ell_k \leftarrow$ each suffix of $P_1$'s intra-tuple separator $S_{1,k-1}$ ($S_{1,K}$ when $k = 1$)
        accept $\ell_k$ if C2 holds of $\ell_k$ and every $\langle P_n, L_n\rangle \in \mathcal{E}$               (ii)
    for $\ell_1 \leftarrow$ each suffix of $P_1$'s head $S_{0,K}$
      for $o \leftarrow$ each substring of $P_1$'s head $S_{0,K}$
        for $c \leftarrow$ each substring of $P_1$'s tail $S_{M,K}$
          accept $o$, $c$ and $\ell_1$ if C4 holds of $o$, $c$ and $\ell_1$ and every $\langle P_n, L_n\rangle \in \mathcal{E}$   (iii)
    return $\langle o, c, \ell_1, r_1, \ldots, \ell_K, r_K\rangle$
```

To see that Generalize$_{\mathrm{OCLR}}$ is consistent, note that the algorithm is correct with respect to the r_k and ℓ_k (for $k > 1$), because **C1** and **C2** are independent. To see that the algorithm is correct with respect to o, c and ℓ_1, note that the algorithm eventually considers all combinations of "sensible" values for each component, just as Generalize$_{\mathrm{HLRT}}$ considers all combinations of sensible values for h, t, and ℓ_1. Thus we have given a proof sketch for the following:

Theorem 5.6 Generalize$_{\mathrm{OCLR}}$ *is consistent.*

We are now in a position to analyze the complexity of Generalize$_{\mathrm{OCLR}}$. As with Generalize$_{\mathrm{HLRT}}$, the bottleneck of the algorithm is line (iii). Line (iii) is executed $\mathrm{O}\left(R^{\frac{5}{3}}\right)$ times, once for each combination of o, c, and ℓ_1, and thus predicate **C4** must be evaluated $\mathrm{O}\left(NR^{\frac{5}{3}}\right)$ times, once for each page. Each such evaluation involves $\mathrm{O}\left(M_{\mathrm{max}}\right)$ primitive string-search operations over strings of length $\mathrm{O}\left(\sqrt[3]{R}\right)$. Therefore, we must execute line (iii)

$$\mathrm{O}\left(NR^{\frac{5}{3}}\right) \times \mathrm{O}\left(M_{\mathrm{max}}\right) \times \mathrm{O}\left(\sqrt[3]{R}\right) = \mathrm{O}\left(M_{\mathrm{max}}NR^2\right)$$

times. Furthermore, lines (i) and (ii) of the algorithm requires time $\mathrm{O}\left(K\right)$. (Since line (iii) dominates the running time, we can safely neglect the dependency on the other parameters.) Thus the total execution time of Generalize$_{\mathrm{OCLR}}$ is bounded by $\mathrm{O}\left(KM_{\mathrm{max}}NR^2\right)$. To summarize:

Theorem 5.7 ($\mathsf{Generalize}_{\mathrm{OCLR}}$ **heuristic-case complexity)**
Under Assumption 4.1, the $\mathsf{Generalize}_{\mathrm{OCLR}}$ *algorithm runs in time* $O\left(KM_{\max}NR^2\right)$.

The HOCLRT **wrapper class.** Learning the LR wrapper class is very simple because all $2K$ components can be learned independently. However, in HLRT and OCLR the choice of ℓ_1 depends on two other components: h and t for HLRT, or o and c for OCLR. As we saw, these dependencies result in the fact that learning HLRT or OCLR is computationally more intensive than learning LR. Since HOCLRT combines the functionality of HLRT and OCLR, learning HOCLRT is in turn computationally more expensive than both HLRT and OCLR.

As usual, we begin with the HOCLRT consistency constraints. HOCLRT wrapper $w = \langle h, t, o, c, \ell_1, r_1, \ldots, \ell_K, r_K\rangle$ is consistent with example $\langle P, L\rangle$ if and only if predicates **C1**, **C2** and **C5** all hold:

$$\begin{aligned}\mathcal{C}_{\mathrm{HOCLRT}}(w, \langle P, L\rangle) \quad \Longleftrightarrow \quad & \bigwedge_{1\le k\le K} \mathbf{C1}(r_k, \langle P, L\rangle) \\ \wedge \quad & \bigwedge_{1< k\le K} \mathbf{C2}(\ell_k, \langle P, L\rangle) \\ \wedge \quad & \mathbf{C5}(h, t, o, c, \ell_1, \langle P, L\rangle).\end{aligned}$$

Predicate **C5** governs the five components h, t, o, c and ℓ_1. **C5** requires that: **(i-ii)** h, t, o and ℓ_1 are satisfactory for the page's head; **(iii)** o, c, and t are satisfactory for the page's tail; and **(iv-v)** o, c, t and ℓ_1 are satisfactory for the page's body. More formally:

$$\begin{aligned}&\mathbf{C5}(h, t, o, c, \ell_1, \langle P, L\rangle) \iff \\ &\qquad ((S_{0,K}/h)/o)/\ell_1 = \ell_1 && \textbf{(i)} \\ &\wedge \quad |((P/h)/o)/t| > |((P/h)/o)/\ell_1| && \textbf{(ii)} \\ &\wedge \quad S_{M,K}/c \ne \Diamond \quad \wedge \quad |(S_{M,K}/c)/o| > |(S_{M,K}/c)/t| && \textbf{(iii)} \\ &\wedge \quad \forall_{1\le m<M} \Big(((S_{m,K}/c)/o)/\ell_1 = \ell_1 && \textbf{(iv)} \\ &\wedge \qquad\qquad |((S^*_{m,K}/c)/o)/t| > |((S^*_{m,K}/c)/o)/\ell_1| \Big). && \textbf{(v)}\end{aligned}$$

As usual, we demand that $\mathcal{C}_{\text{HOCLRT}}$ be correct:

Theorem 5.8 ($\mathcal{C}_{\text{HOCLRT}}$ is correct.) *For every* HOCLRT *wrapper w, page P, and label L, $\mathcal{C}_{\text{HOCLRT}}(w, \langle P, L\rangle) \iff \mathsf{ExecHOCLRT}(w, P) = L$.*

The proof is omitted; it is similar to that of Theorems 4.1, 5.2 and 5.5.

We are now in a position to state the HOCLRT generalization function:

```
Generalize_HOCLRT(examples $\mathcal{E} = \{\langle P_1, L_1\rangle, \ldots, \langle P_N, L_N\rangle\}$)
  for each $1 \le k \le K$
    for $r_k \leftarrow$ each prefix of $P_1$'s intra-tuple separator for $S_{1,k}$
      accept $r_k$ if C1 holds of $r_k$ and every $\langle P_n, L_n\rangle \in \mathcal{E}$
  for each $1 < k \le K$
    for $\ell_k \leftarrow$ each suffix of $P_1$'s intra-tuple separator $S_{1,k-1}$ (or $S_{1,K}$ when $k = 1$)
      accept $\ell_k$ if C2 holds of $\ell_k$ and every $\langle P_n, L_n\rangle \in \mathcal{E}$
  for $\ell_1 \leftarrow$ each suffix of $P_1$'s head $S_{0,K}$
    for $o \leftarrow$ each substring of $P_1$'s head $S_{0,K}$
      for $c \leftarrow$ each substring of $P_1$'s tail $S_{M,K}$
        for $h \leftarrow$ each substring of $P_1$'s head $S_{0,K}$
          for $t \leftarrow$ each substring of $P_1$'s tail $S_{M,K}$
            accept $h$, $t$, $o$, $c$ and $\ell_1$ if C5 holds of them and every $\langle P_n, L_n\rangle \in \mathcal{E}$     (i)
  return $\langle h, t, o, c, \ell_1, r_1, \ldots, \ell_K, r_K\rangle$
```

The algorithm $\mathsf{Generalize}_{\text{HOCLRT}}$ follows the same pattern as established by HLRT and OCLR: the values of r_k and ℓ_K (for $k < 1$) can be learned independently, but line (i) indicates that values for the remaining components (h, t, o, c, and ℓ_1 in this case) can be selected only by enumerating all combinations of candidates for each.

As with HLRT and OCLR, a complexity analysis of $\mathsf{Generalize}_{\text{HOCLRT}}$ reveals that line (i) is the bottleneck. Specifically, we must test predicate **C5** against $\mathrm{O}\left(\sqrt[3]{R}\right) \times \mathrm{O}\left(R^{\frac{2}{3}}\right) \times \mathrm{O}\left(R^{\frac{2}{3}}\right) \times \mathrm{O}\left(R^{\frac{2}{3}}\right) \times \mathrm{O}\left(R^{\frac{2}{3}}\right) = \mathrm{O}\left(R^3\right)$ different combinations of h, t, o, c and ℓ_1. Each evaluation of predicate **C5** involves $\mathrm{O}\left(NM_{\text{max}}\right)$ invocations of the primitive string search operation, which has cost $\mathrm{O}\left(\sqrt[3]{R}\right)$. Furthermore, the rest of the algorithm requires time $\mathrm{O}\left(K\right)$. Combining these bounds, we have that the total running time is:

$$\mathrm{O}\left(K\right) \times \mathrm{O}\left(N M_{\mathrm{max}}\right) \times \mathrm{O}\left(R^{3}\right) \times \mathrm{O}\left(\sqrt[3]{R}\right) = \mathrm{O}\left(K N M_{\mathrm{max}} R^{\frac{10}{3}}\right).$$

To summarize:

> **Theorem 5.9** ($\mathsf{Generalize}_{\mathrm{HOCLRT}}$ **heuristic-case complexity)**
> *Under Assumption 4.1, the* $\mathsf{Generalize}_{\mathrm{HOCLRT}}$ *algorithm runs in time* $O\left(K N M_{\mathrm{max}} R^{\frac{10}{3}}\right)$.

5.3 Nested resources

The preceding section was concerned with alternatives to HLRT for information resources that render data in a tabular format. In this section we explore one particular style of non-tabular formatting, which we call *nested* structure.

While a rectangular table is the prototypical example of a document exhibiting tabular structure, a "table of contents" is the prototype of nested structure:

```
Part I: Introduction to CMOS Technology
   Chapter 1: Introduction to CMOS Circuits
      1.1   A Brief History
      1.2   MOS Transistors
      1.3   CMOS Logic
         1.3.1   The Inverter
         1.3.2   Combinatorial Logic
      1.4   Circuit and System Representation
         1.4.1   Behavioral Representation
         1.4.2   Structural Representation
      1.5   Summary
   Chapter 2: MOS Transistor Theory
      2.1   Introduction
         2.1.1   nMOS Enhancement Transistor
         2.1.2   pMOS Enhancement Transistor
      2.2   MOS Device Design Equations
         2.2.1   Basic DC Equations
         2.2.2   Second Order Effects
            2.2.2.1   Threshold Voltage—Body Effect
            2.2.2.2   Subthreshold Region
```

2.2.2.3 Channel Length Modulation
2.2.3 MOS Models
2.3 The Complementary CMOS Inverter—DC Characteristics
⋮
Part II: System Design and Design Methods
Chapter 6: CMOS Design Methods
6.1 Introduction
⋮

In a document with nested structure, values for a set of K attributes are presented, with the information organized hierarchically. The information residing "below" attribute k can be thought of as details about attribute k. Moreover, for each attribute, there may be any number (zero, one, or more) of values for the given attribute. The only constraint is that values can be provided for attribute k only if values are also provided for attributes 1 to $k-1$.

In the earlier table-of-contents example, there are $K = 5$ attributes: the titles of the book's parts, chapters, sections, subsections, and sub-subsections. The constraint that each attribute can have any number of values corresponds to the idea that a book can have any number of parts, a part can have any number of chapters, a chapter can have any number of sections, and so on. The constraint that attribute k can have a value only if attributes 1 to $k-1$ have values means that there can be no "floating" sub-subsection without an enclosing subsection, no "floating" subsections without an enclosing section, and so forth.

With a tabular format, there is a natural definition of a page's information content; namely, the substrings of the source document corresponding to the attribute values for each tuple. For nested documents, we extend this idea in a straightforward manner. The information content of a nested document is a tree of depth of most K. The nodes of the tree group together related attribute values, while the edges encode the attribute values themselves.

For example, the information content of the table-of-contents example is repre-

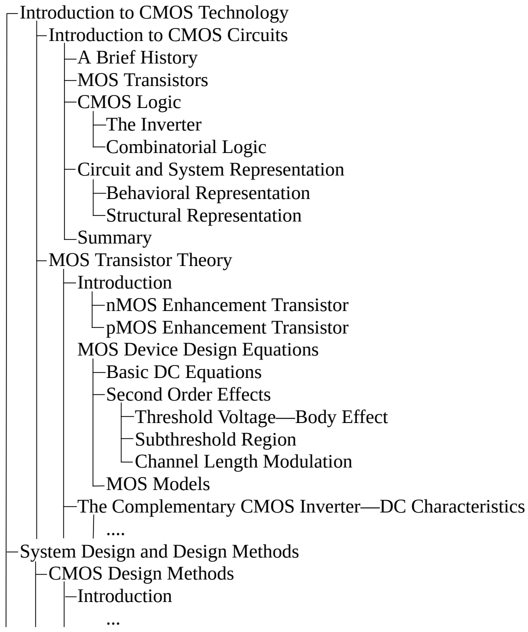

Introduction to CMOS Technology
Introduction to CMOS Circuits
A Brief History
MOS Transistors
CMOS Logic
The Inverter
Combinatorial Logic
Circuit and System Representation
Behavioral Representation
Structural Representation
Summary
MOS Transistor Theory
Introduction
nMOS Enhancement Transistor
pMOS Enhancement Transistor
MOS Device Design Equations
Basic DC Equations
Second Order Effects
Threshold Voltage—Body Effect
Subthreshold Region
Channel Length Modulation
MOS Models
The Complementary CMOS Inverter—DC Characteristics
....
System Design and Design Methods
CMOS Design Methods
Introduction
...

Figure 5.2: *An example of a nested documents information content.*

sented as following tree shown in Figure 5.2. (The ellipses indicate parts of the tree that were omitted in the example; naturally, the information content of an actual document does not contain ellipses.)

Note that tabular formatting is actually a special case of a nested formatting, in which the tree's root has zero or more children, all interior nodes have exactly one child, and every leaf is at depth exactly K.

For tabular formatting, we formalized the notion of information content in terms of a label L of the form

$$L = \left\{ \begin{array}{c} \left\langle \langle b_{1,1}, e_{1,1} \rangle, \ldots, \langle b_{1,k}, e_{1,k} \rangle, \ldots, \langle b_{1,K}, e_{1,K} \rangle \right\rangle \\ \vdots \\ \left\langle \langle b_{M,1}, e_{M,1} \rangle, \ldots, \langle b_{M,k}, e_{M,k} \rangle, \ldots, \langle b_{M,K}, e_{M,K} \rangle \right\rangle \end{array} \right\}.$$

To precisely discuss wrappers for documents with nested structure, we must generalize this formalization of a label.

Since we must represent trees with arbitrary depth, we employ a recursive definition to describe a nested-structure label L:

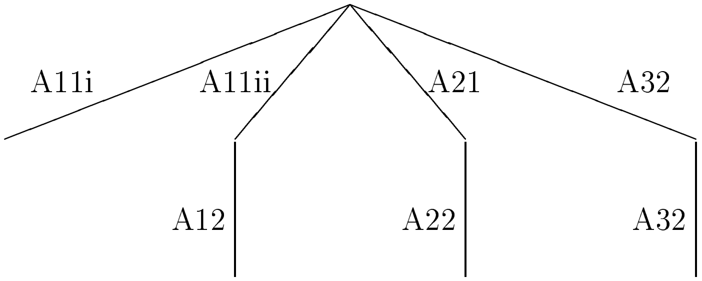
$$\begin{array}{rcl} L & \Rightarrow & \mathsf{label} \\ \mathsf{label} & \Rightarrow & \{\ldots, \mathsf{node}, \ldots\} \\ \mathsf{node} & \Rightarrow & \langle\langle b, e\rangle, \mathsf{label}\rangle. \end{array}$$

That is, L is a label structure, where a label is a set of zero or more node structures, and a node is a pair consisting of an interval $\langle b, e\rangle$ and a label structure. A node's interval $\langle b, e\rangle$ indicates the indices of the string used to label the edge between the node and its parent. A node's label represents its children.

For example, consider the following example page:

`h[A11i][A11ii](A12)[A21](A22)[A31](A32)t`

The task is to extract the following nested structure:



The following label structure L represents this tree:

$$L = \left\{ \begin{array}{c} \langle\langle 3, 6\rangle, \{\}\rangle\,, \\ \langle\langle 9, 13\rangle, \{\langle\langle 16, 18\rangle, \{\}\rangle\}\rangle\,, \\ \langle\langle 21, 23\rangle, \{\langle\langle 26, 28\rangle, \{\}\rangle\}\rangle\,, \\ \langle\langle 31, 33\rangle, \{\langle\langle 36, 38\rangle, \{\}\rangle\}\rangle \end{array} \right\}$$

For instance, `A22` is substring $[26, 28]$ of the example page. Since `A22` is the value of the second attribute under the third value of the first attribute, the pair $[26, 28]$ occupies the corresponding location in the label.

Recall that when we defined a tabular label, we placed constraints on indices which ensure that the layout is in fact tabular; see Footnote 1 on page 24. A similar requirement must be satisfied by the indices in a nested-structure label. Specifically, the indices must in fact correspond to a tree-like decomposition of the original document.

Finally, note that since tabular formatting is a special case of nested formatting, we will overload the symbol L to mean either a tabular or a nested label; the interpretation will be clear from context.

5.3.1 The N-LR *and* N-HLRT *wrapper classes*

In this section we elaborate on the idea of nested structure by developing wrapper classes, N-LR and N-HLRT, that can be used to extract such structure. Both wrapper classes are similar to the tabular wrapper classes: associated with each of the K attributes is a left-hand delimiter ℓ_k and a right-hand delimiter r_k. In the tabular wrappers, after extracting the value of the $k^{\underline{\text{th}}}$ attribute for some particular tuple, then the wrapper extracts the $(k+1)^{\underline{\text{st}}}$ attribute value (or the first, in the case of $k = K$). Nested-structure wrappers generalize this procedure: after extracting the $k^{\underline{\text{th}}}$ attribute, the next extracted value could belong to attributes $k+1$, k, $k-1$, $k-2$, $\ldots$, 2 or 1. The N-LR and N-HLRT wrappers use the position of the ℓ_k delimiters to indicate how the page should be interpreted: the next value to be extracted is indicated by which ℓ_k delimiters occurs next. For example, if we have currently extracted a page's content through index 1350 and there is a subsection title starting at position 1450, a chapter title starting at position 1400, and a sub-subsection title starting at position 1500, then we will extract the chapter title next.

The N-LR wrapper class. The N-LR wrapper class is very similar to the LR wrapper class, except that the execution of the wrapper searchers for the delimiters in a different way. Specifically, after extracting a value for the $k^{\underline{\text{th}}}$ attribute, the ExecN-LR determines which attribute occurs next. As just mentioned, after attribute k, the next attribute could be at attributes 1 to $k+1$. (Of course, at attribute K the next attribute can not be at $k = K+1$, and the first extracted attribute must be at $k = 1$.) To determine which attribute occurs next, the positions of the left delimiter $\ell_{k'}$ are compared, for each possible next attributes k'.

```
ExecN-LR(wrapper $\langle \ell_1, r_1, \ldots, \ell_K, r_K \rangle$, page $P$)
    $i \leftarrow 0$
    $k \leftarrow 0$
    loop
        $k \leftarrow \arg\min_{k' \in [1, \min(1+k, K)]} P[i] \# \ell_{k'}$      (i)
        exit loop if no such $k$ exists                                         (ii)
        $i \leftarrow i + P[i] \# \ell_k + |\ell_k|$
        $b \leftarrow i$
        $i \leftarrow i + P[i] \# r_k$
        $e \leftarrow i - 1$
        insert $\langle b, e \rangle$ as indices of next value of $k^{\underline{\text{th}}}$ attribute
    return label structure
```

Line (i) is the key to the operation of ExecN-LR. The algorithm determines which (among the $k+1$ possibilities) attribute k' occurs next, by finding the minimum value of $P[i]\#\ell_{k'}$. If line (i) were replaced by "$k \leftarrow$ if $k = K$ then 1 else $1+k$", then ExecN-LR would reduce to ExecLR. The termination condition (line (ii)) is satisfied whenever none of the $\ell_{k'}$ occur—*i.e.*, when the wrapper has reached the end of the page.

We illustrate N-LR with two simple examples. First, recall the artificial example `ho[A11](A12)co[A21](A22)co[A31](A32)ct`. The N-LR wrapper $\langle \texttt{[}, \texttt{]}, \texttt{(}, \texttt{)} \rangle$ is consistent with this example page.

Of course, this example is tabular rather than nested. This example illustrates that tabular formatting is a special case of nested formatting. To illustrate the additional functionality of N-LR, recall the example page introduced earlier,

```
h[A11i][A11ii](A12)[A21](A22)[A31](A32)t
```

The N-LR wrapper $\langle \texttt{[}, \texttt{]}, \texttt{(}, \texttt{)} \rangle$ correctly extracts this structure from the example page.

The N-HLRT wrapper class. The N-HLRT wrapper class combines the functionality of N-LR and HLRT. The execution of an N-HLRT wrapper $\langle h, t, \ell_1, r_1, \ldots, \ell_K, r_K \rangle$ is defined as follows.

```
ExecN-HLRT(wrapper $\langle h, t, \ell_1, r_1, \ldots, \ell_K, r_K \rangle$, page $P$)
    $i \leftarrow P\#h$
    $k \leftarrow 0$
    loop
        $k \leftarrow \arg\min_{k' \in [1, \min(1+k, K)]} P[i]\#\ell_{k'}$
        exit loop if no such $k$ exists or $|P[i]/\ell_k| \geq |P[i]/t|$
        $i \leftarrow i + P[i]\#\ell_k + |\ell_k|$
        $b \leftarrow i$
        $i \leftarrow i + P[i]\#r_k$
        $e \leftarrow i - 1$
        insert $\langle b, e \rangle$ as indices of next value of $k^{\underline{\text{th}}}$ attribute
    return label structure
```

This procedure operates just like ExecN-LR except that additional mechanisms (similar to those in ExecHLRT) are needed to handle the h and t components.

As expected, the N-HLRT wrapper $\langle \texttt{h}, \texttt{t}, \texttt{[}, \texttt{]}, \texttt{(}, \texttt{)} \rangle$ is consistent with the example pages mentioned earlier,

```
ho[A11](A12)co[A21](A22)co[A31](A32)ct
```

and

```
h[A11i][A11ii](A12)[A21](A22)[A31](A32)t
```

5.3.2 Relative expressiveness

As with the four tabular wrapper classes, we are interested in comparing the relative expressiveness of N-LR and N-HLRT. With six wrapper classes, there are potentially 2^6 regions that must be examined in a Venn diagram (such as Figure 5.1) illustrating the relative expressiveness of the classes. Therefore, for brevity we will compare only LR, HLRT, N-LR and N-HLRT.

Figure 5.3 and Theorem 5.10 capture our results concerning the relative expressiveness of these four wrapper classes.

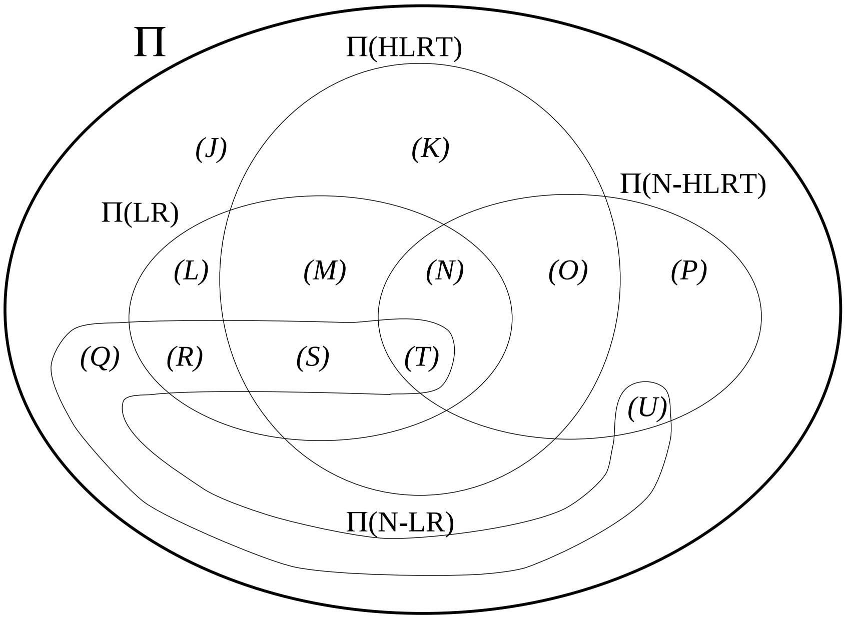


Figure 5.3: *The relative expressivenes of the* LR*,* HLRT*,* N-LR*, and* N-HLRT *wrapper classes.*

> **Theorem 5.10 (Relative expressiveness of** LR**,** HLRT**,** N-LR **and** N-HLRT**)** *The relationships between* Π(LR)*,* Π(HLRT)*,* Π(N-LR) *and* Π(N-HLRT) *are as indicated in Figure 5.3.*

Proof of Theorem 5.10 (Sketch): To see that these relationships hold, it suffices to show that:

1. There exists at least one pair $\langle P, L\rangle \in \Pi$ in each of the regions marked *(J)*, *(K)*, *(L)*, …, *(U)* in Figure 5.3;
2. Every pair in N-LR and HLRT is also in LR: $(\Pi(\text{N-LR}) \cap \Pi(\text{HLRT})) \subset \Pi(\text{LR})$.
3. Every pair in N-HLRT and LR is also in HLRT: $(\Pi(\text{N-HLRT}) \cap \Pi(\text{LR})) \subset \Pi(\text{HLRT})$.

Note that these three assertions jointly imply that the four wrapper classes are related as claimed.

Consider each assertion in turn.

1. In Appendix B.6 we identify one $\langle P, L\rangle$ pair in each of the regions *(J)*, *(K)*, *etc.*

2. Consider a pair $\langle P, L\rangle$ claimed to be a member of N-LR and HLRT but not LR. Since $\langle P, L\rangle \in \Pi(\text{HLRT})$, $\langle P, L\rangle$ must have a tabular structure. It is straightforward to show that if $\langle P, L\rangle \in \Pi(\text{N-LR})$ and $\langle P, L\rangle$ is tabular, then $\langle P, L\rangle \in \Pi(\text{LR})$. But this contradicts the assumption that $\langle P, L\rangle \notin \Pi(\text{LR})$.
3. Consider a pair $\langle P, L\rangle$ claimed to be a member of LR and N-HLRT but not HLRT. Since $\langle P, L\rangle \in \Pi(\text{LR})$, $\langle P, L\rangle$ must have a tabular structure. It is straightforward to show that if $\langle P, L\rangle \in \Pi(\text{N-HLRT})$ and $\langle P, L\rangle$ is tabular, then $\langle P, L\rangle \in \Pi(\text{HLRT})$. But this contradicts the assumption that $\langle P, L\rangle \notin \Pi(\text{HLRT})$.

See Appendix B.6 for details.

□ (Proof of Theorem 5.10 (Sketch))

5.3.3 Complexity of learning

To perform a complexity analysis of learning LR, HLRT, OCLR and HOCLRT wrappers, we stated the consistency constraints that must hold if a wrapper is to be consistent with a particular example. We then described the Induce generalization function for a particular wrapper class as a special-purpose constraint-satisfier for the corresponding consistency constraints. More precisely, when generalizing from a set $\mathcal{E}$ of examples, the consistency constraint $\mathcal{C}_{\mathcal{W}}$ for wrapper class $\mathcal{W}$ tells us whether some wrapper $w \in \mathcal{H}_{\mathcal{W}}$ obeys $w(P) = L$ for each $\langle P, L\rangle \in \mathcal{E}$.

An alternative is to use the wrapper execution procedure Exec$\mathcal{W}$ directly. For instance, rather than testing whether some wrapper $w \in \mathcal{H}_{\text{HLRT}}$ obeys $\mathcal{C}_{\text{HLRT}}$, we could have used the ExecHLRT procedure to verify that w is consistent with each example.

The reason we did not pursue this path is computational: by explicitly stating the constraint $\mathcal{C}_{\text{HLRT}}$, we can then decompose the constraint into **C1**–**C3**, show that these predicates are independent, and thereby search for consistent wrappers much more efficiently. For the HLRT class, the original inefficient Generalize$_{\text{HLRT}}$ algorithm tries to satisfy $\mathcal{C}_{\text{HLRT}}$ directly, while the improved Generalize$^*_{\text{HLRT}}$ algorithm exploits the independence of **C1**–**C3**. A similar strategy was applied to the LR, OCLR, and HOCLRT classes.

Two problems arise when applying this methodology to the N-LR and N-HLRT wrapper classes. First of all, the consistency constraints $\mathcal{C}_{\text{N-LR}}$ and $\mathcal{C}_{\text{N-HLRT}}$ would be quite complicated, requiring substantial additional notation beyond the variables $S_{m,K}$ and $A_{m,k}$.

Further consideration of this first difficulty leads to the second problem. To understand why these constraints are so complicated, consider attribute k. What values are satisfactory for ℓ_k (the left-hand delimiter for the $k^{\underline{\text{th}}}$ attribute)? Recall that after extracting a value for the $k^{\underline{\text{th}}}$ attribute, ExecN-LR and ExecN-HLRT might extract a value for attribute $k+1$, k, $k-1$, …, 2, or 1, depending on which value $\ell_{k'}$ occurs first amongst the $k' \in [1, k+1]$. Thus ℓ_k *must occur* first when attribute k should be extracted, but it *must not occur* first when some other attribute $k+1$, $k-1$, $k-2$ …, 2, or 1 are to be extracted. To summarize, whether a candidate value for ℓ_k is satisfactory depends on the choices of up to k other left-hand delimiters.[4]

This discussion reveals the second problem with trying to decompose the consistency constraints $\mathcal{C}_{\text{N-LR}}$ or $\mathcal{C}_{\text{N-HLRT}}$: we conjecture that these constraints simply can not be decomposed in a way that simplifies learning wrappers from these classes. Since we can see no advantage in explicitly stating the constraints $\mathcal{C}_{\text{N-LR}}$ and $\mathcal{C}_{\text{N-HLRT}}$, we will not do so. Fortunately, once we've committed to this decision, the analysis of the complexity of learning N-LR and N-HLRT is greatly simplified.

The N-LR wrapper class. The previous discussion suggests the following N-LR generalization function:

[4] Throughout this discussion we ignore the fact that attribute K is special case, because this complication does not greatly change the computational burden of learning N-LR and N-HLRT wrappers.

```
Generalize$_{\text{N-LR}}$(examples $\mathcal{E} = \{\langle P_1, L_1\rangle, \ldots, \langle P_N, L_N\rangle\}$)
    for $r_1 \leftarrow$ each prefix of the text following some instance of attribute 1
        $\ddots$
        for $r_K \leftarrow$ each prefix of the text following some instance of attribute $K$
            for $\ell_1 \leftarrow$ each suffix of the text preceding some instance of attribute 1
                $\ddots$
                for $\ell_K \leftarrow$ each suffix of the text preceding some instance of attribute $K$
                    $w \leftarrow \langle \ell_1, r_1, \ldots, \ell_K, r_K\rangle$
                    if ExecN-LR$(w, P_n) = L_n$ for every $\langle P_n, L_n\rangle \in \mathcal{E}$, then      (i)
                        return $w$
```

As indicated, note that line (i) of the algorithm simply uses the ExecN-LR procedure to determine whether a candidate wrapper is consistent. The following theorem follows immediately:

Theorem 5.11 Generalize$_{\text{N-LR}}$ *is consistent.*

Before proceeding with the complexity analysis, let us describe Generalize$_{\text{N-LR}}$ in more detail. Note that the candidates for the r_k and ℓ_k are specified differently from Generalize$_{\text{N-LR}}$ than for the four wrapper classes discussed earlier. In Generalize$_{\text{LR}}$, for example, the candidates for r_k are the prefixes of the intra-tuple separator for $S_{1,k}$ for page P_1. However, we have not yet defined the notation $S_{m,k}$ for N-LR. (The reason for not defining $S_{m,k}$ for N-LR was just mentioned: there is no advantage—either computational or to simplify the presentation—to explicitly stating the consistency constraint $\mathcal{C}_{\text{N-LR}}$ and thus there was no need to define $S_{m,k}$ and $A_{m,k}$ for nested resources.) Rather than invent a formal notation, we will describe the candidates informally. For example, the candidates for r_k in the N-LR and LR classes are similar: we look for some occurrence of the $k^{\underline{\text{th}}}$ attribute, and then the candidates for r_k are the prefixes of the text between this attribute and the next. It does not matter which such occurrence is considered, though for efficiency we assume that the algorithm considers the occurrence that minimizes the number of candidates for r_k.

We now provide a complexity analysis for the $\mathsf{Generalize}_{\text{N-LR}}$ algorithm. As usual, there are $\mathrm{O}\left(\sqrt[3]{R}\right)$ candidates for each of r_k and ℓ_k, and thus we must evaluate line (i) $\mathrm{O}\left(R^{\frac{2K}{3}}\right)$ times. Each execution of line (i) required N invocations of the ExecN-LR function. ExecN-LR runs in time linear in the length of the page. Let $R_{\max} = \max_n |P_n|$ be the length of the longest example page. Combining these bounds, we have that the total running time of $\mathsf{Generalize}_{\text{N-LR}}$ is bounded by $\mathrm{O}\left(R^{\frac{2K}{3}}\right) \times \mathrm{O}(N) \times \mathrm{O}(R_{\max}) = \mathrm{O}\left(R^{\frac{2K}{3}} N R_{\max}\right)$. To simplify this expression at the cost of a somewhat looser bound, we can make use of the fact that $R \leq R_{\max}$, and thus $\mathrm{O}\left(R^{\frac{2K}{3}} N R_{\max}\right) = \mathrm{O}\left(N R_{\max}^{1+\frac{2K}{3}}\right) = \mathrm{O}\left(N R_{\max}^{\frac{2K+3}{3}}\right)$. Thus we have a proof for the following theorem:

Theorem 5.12 ($\mathsf{Generalize}_{\text{N-LR}}$ heuristic-case complexity)
Under Assumption 4.1, the $\mathsf{Generalize}_{\text{N-LR}}$ algorithm runs in time $O\left(N R_{\max}^{\frac{2K+3}{3}}\right)$.

The N-HLRT wrapper class. We just observed that a discussion of the disadvantages of explicitly stating $\mathcal{C}_{\text{N-LR}}$ resulted in the straightforward $\mathsf{Generalize}_{\text{N-LR}}$ algorithm. Similarly, there is no point in explicitly stating the consistency constraint $\mathcal{C}_{\text{N-HLRT}}$, and thus the generalization function $\mathsf{Generalize}_{\text{N-HLRT}}$ is also rather straightforward:

```
Generalize_N-HLRT(examples $\mathcal{E} = \{\langle P_1, L_1\rangle, \ldots, \langle P_N, L_N\rangle\}$)
  for $r_1 \leftarrow$ each prefix of the text following some instance of attribute 1
    $\ddots$
      for $r_K \leftarrow$ each prefix of the text following some instance of attribute $K$
        for $\ell_1 \leftarrow$ each suffix of the text preceding some instance of attribute 1
          $\ddots$
            for $\ell_K \leftarrow$ each suffix of the text preceding some instance of attribute $K$
              for $h \leftarrow$ each substring of the text preceding the first attribute
                for $t \leftarrow$ each substring of the text following the last attribute
                  $w \leftarrow \langle h, t, \ell_1, r_1, \ldots, \ell_K, r_K\rangle$
                  if ExecN-HLRT$(w, P_n) = L_n$ for every $\langle P_n, L_n\rangle \in \mathcal{E}$, then      (i)
                    return $w$
```

The difference between Generalize$_{\text{N-HLRT}}$ and Generalize$_{\text{HLRT}}$ is similar to the difference between Generalize$_{\text{LR}}$ and Generalize$_{\text{N-LR}}$. The main difference is that line (i) of the algorithm uses ExecN-HLRT rather than $\mathcal{C}_{\text{N-HLRT}}$ to verify wrappers.

As usual, we begin by observing that Generalize$_{\text{N-HLRT}}$ behaves correctly (like that of Theorem 5.11, the proof follows immediately from line (i).)

Theorem 5.13 Generalize$_{\text{N-LR}}$ *is consistent.*

The complexity analysis of Generalize$_{\text{N-HLRT}}$ follows that of Generalize$_{\text{N-LR}}$. The difference is that we must invoke line (i) of the algorithm $\mathrm{O}\left(R^{\frac{2K+4}{3}}\right)$ times rather than $\mathrm{O}\left(R^{\frac{2K}{3}}\right)$ times. (The extra "4" in the exponent arises from the fact that we must examine $\mathrm{O}\left(R^{\frac{2}{3}}\right)$ candidates for each of h and t.) From this point the analysis proceeds as before, and we have that the complexity of Generalize$_{\text{N-HLRT}}$ $\mathrm{O}\left(R^{\frac{2K+4}{3}} NR_{\max}\right) = \mathrm{O}\left(NR_{\max}^{1+\frac{2K+4}{3}}\right) = \mathrm{O}\left(NR_{\max}^{\frac{2K+7}{3}}\right)$. We can summarize these observations with the following theorem.

Theorem 5.14 (Generalize$_{\text{N-HLRT}}$ heuristic-case complexity) *Under Assumption 4.1, the* Generalize$_{\text{N-HLRT}}$ *algorithm runs in time* $O\left(NR_{\max}^{\frac{2K+7}{3}}\right)$.

5.4 Summary

In this chapter, we have described several wrapper classes. First, we examined the LR, OCLR and HOCLRT wrapper classes. Like HLRT, these three classes are designed for extracting data from resources exhibiting a tabular structure. Second, we examined two wrapper classes—N-LR and N-HLRT—that are designed for extracting information from resources that exhibit a more general nested structure.

Our formal results are summarized in the following table:

wrapper class	heuristic-case learning complexity	strictly more expressive than
LR	$\mathrm{O}\left(KM_{\max}NR^{\frac{2}{3}}\right)$	×
HLRT	$\mathrm{O}\left(KM_{\max}NR^{2}\right)$	×
OCLR	$\mathrm{O}\left(KM_{\max}NR^{2}\right)$	LR
HOCLRT	$\mathrm{O}\left(KNM_{\max}R^{\frac{10}{3}}\right)$	HLRT
N-LR	$\mathrm{O}\left(NR_{\max}^{\frac{2K+3}{3}}\right)$	×
N-HLRT	$\mathrm{O}\left(NR_{\max}^{\frac{2K+7}{3}}\right)$	×

The first column indicates the six wrapper classes we have investigated:

LR, the simplest wrapper class for tabular resources, consisting of just left- and right-hand attribute delimiters;

HLRT, the wrapper class discussed extensively in Chapter 4;

OCLR, a wrapper class consisting of left- and right-hand attribute delimiters as well as an opening and closing delimiter for each tuple;

HOCLRT, a wrapper class that combined the functionality of HLRT and OCLR;

N-LR, the simplest wrapper class for nested resources, consisting of just left- and right-hand attribute delimiters; and

N-HLRT, the straightforward extension of HLRT to nested resources.

The second column in the table provides a way to compare the difficulty of automatically earning wrappers within each class: we have analyzed the heuristic-case (defined by Assumption 4.1) running-time of the function $\mathsf{Generalize}_{\mathcal{W}}$, for each wrapper class $\mathcal{W}$. The analysis in provided in terms of the following parameters, which characterize the set of examples:

K, the number of attributes per tuple;
N, the number of examples;
M_{tot}, the total number of tuples in the example pages;
R, the length of the shortest example page; and
R_{max}, the length of the longest example page.

The third column in the table provides a brief summary of the relative expressiveness of the six wrapper classes. Relative expressiveness captures the extent to which wrappers from one class can mimic the behavior of wrappers from another. As Theorems 5.1 and 5.10 and Figures 5.1 and 5.3 indicate, this summary is very crude indeed; the full analysis reveals that the classes are interrelated in a substantially richer manner.

These theoretical results are important and interesting, but of course the bottom line is the extent to which these six wrapper classes are useful for real information resources. In Section 7.2, we demonstrate that indeed these wrapper classes are quite useful: collectively they can handle 70% of a recently-surveyed sample of actual Internet resources, with the individual classes handling between 13% and 57%.

Chapter 6

CORROBORATION

6.1 Introduction

Our discussion of automatic wrapper construction has focused on inductive learning. One input to the Induce generic learning algorithm is the oracle function $\mathsf{Oracle}_{\mathcal{T},\mathcal{D}}$ (See Figure 3.3). The oracle supplies Induce with a stream of labeled examples from the resource under consideration. So far, we have taken $\mathsf{Oracle}_{\mathcal{T},\mathcal{D}}$ to be provided as input. And indeed, this *supervised* approach is standard practice in many applications of machine learning.

Unfortunately, assuming $\mathsf{Oracle}_{\mathcal{T},\mathcal{D}}$ as an input usually means that a *person* must play the role of the oracle, painstakingly gathering and examining a collection of example query responses. Since our goal is *automatic* wrapper construction, requiring a person to be "in the loop" in this way is regrettable.

Therefore, in this chapter we focus on automating the $\mathsf{Oracle}_{\mathcal{T},\mathcal{D}}$ function. (Throughout, we discuss only the HLRT wrapper class; analogous results could be obtained for the other wrapper classes.) Conceptually, oracles perform two functions: gathering pages, and labeling them. Our effort is directed only at the second step (see [Doorenbos et al. 97] for work on the first). We introduce *corroboration*, a technique for automatically labeling a page. Though in this thesis we do not reach our goal of *entirely* automatic wrapper construction, we think that corroboration represents a significant step in that direction.

Corroboration assumes a library of domain-specific attribute *recognizers*. Each recognizer reports the instances present in the page for a particular attribute. For example, one recognizer might tell the corroboration system the location of all the countries, while a second recognizer locates the country codes. Corroboration is the process of sifting through the evidence provided by the recognizers.

If the recognizers do not make mistakes, then corroboration is very simple. Most of this chapter is concerned with the complications that arise due to the recognizers' mistakes. However, we note that even if the recognizers were perfect, wrapper induction would still be important, because the recognizers might run slowly, while wrappers are used on-line and therefore should be very fast.

We begin with a high-level description of the issues related to corroboration (Section 6.2). We then provide a formal specification of the corroboration process (Section 6.3). Third, we describe an algorithm, Corrob, that solves an interesting special case of the general problem (Section 6.4). Fourth, we show how to use the output of Corrob to implement the $\mathsf{Oracle}_{\mathcal{T},\mathcal{D}}$ function; there turn out to be several wrinkles (Section 6.5). While clean and elegant, Corrob is extremely slow, and so in Section 6.6 we describe several heuristics and simplifications that result in a usable algorithm, called Corrob*. Finally, we step back and discuss the concept of recognizers, asking whether they represent a realistic approach to labeling pages (Section 6.7).

6.2 The issues

We begin with an example that illustrates at a fairly high level the basic issues that are involved. Consider a simple extension of the country/code example (Figure 2.1) that has three (rather than two) attributes: country, country code, and capital city. In this section we will abbreviate these attributes CTRY, CODE and CAP, respectively.

Recognizers. The *labeling problem* under consideration is to take as input a page from the country/code/capital resource, and to produce as output a label for the page. In this chapter we explore one strategy for automatically solving the labeling problem. We assume we have a library of *recognizers*, one for each attribute. A recognizer examines the page to be labeled, and identifies all instances of a particular attribute. For example, a recognizer for the CTRY attribute would scan a page and return a list of the page fragments that the recognizer "believes" to be values of the CTRY attribute.

More formally, a recognizer is a function taking as input a page, and producing as output a set of subsequences of the page. Each such subsequence is called an *instance* of the recognized attribute. The symbol $\mathcal{R}$ indicates a recognizer; for example, $\mathcal{R}_{\text{CTRY}}$ is the country recognizer.

Note that the recognizer library does *not* specify the order in which the attributes are rendered by the information resource under consideration. Our intent is to make the wrapper construction system as general as possible. For example, ideally we would like to provide our system with example pages from *any* site that provides country/code/capital information; we want the system to learn that at one resource the tuples are formatted as $\langle \text{CTRY}, \text{CODE}, \text{CAP} \rangle$, while at another they are formatted as $\langle \text{CODE}, \text{CAP}, \text{CTRY} \rangle$.

Corroborating perfect recognizers. As mentioned, $\mathcal{R}_{\text{CTRY}}$ is a recognizer for countries, $\mathcal{R}_{\text{CODE}}$ recognizes country codes, and so forth. Ideally, each recognizer will behave *perfectly*: every instance returned by the recognizer is in fact a true instance of the attribute, and the recognizer never fails to output an instance of an attribute that is present in the input.

If every recognizer is perfect, then the labeling problem is trivial: we can simply compute the (unique) attribute ordering consistent with the recognized instances, and then gather the instances into an overall label for the page.

For example, suppose that all three recognizers $\mathcal{R}_{\text{CTRY}}$, $\mathcal{R}_{\text{CODE}}$ and $\mathcal{R}_{\text{CAP}}$ are perfect. And suppose that when we apply these recognizes to some page, we obtain the following output:

$\mathcal{R}_{\text{CAP}}$ *perfect*
$\langle 29, 34\rangle$
$\langle 49, 54\rangle$
$\langle 69, 74\rangle$

$\mathcal{R}_{\text{CTRY}}$ *perfect*
$\langle 15, 20\rangle$
$\langle 35, 40\rangle$
$\langle 55, 60\rangle$

$\mathcal{R}_{\text{CODE}}$ *perfect*
$\langle 22, 27\rangle$
$\langle 42, 47\rangle$
$\langle 62, 67\rangle$

First of all, note that the indices in this table are completely artificial; they bear no resemblance to the HTML code displayed in Figure 2.1.

Each table contains a set of indices, each of the form $\langle b, e\rangle$, indicating the beginning and end indices (respectively) of the instance. The second table above, for example, indicates that there are country attributes in the input page at positions 15–20, 35–40, and 55–60.

Since each recognizer is perfect, note that there is a unique ordering consistent with the recognized instances. In this case, the ordering is $\langle \text{CTRY}, \text{CODE}, \text{CAP}\rangle$.

Given this ordering, it is now trivial to compose the sets of recognized instances into a label for the overall page. In this example, we obtain the following label:

$$\left\{ \begin{array}{l} \langle\langle 15, 20\rangle, \langle 22, 27\rangle, \langle 29, 34\rangle\rangle, \\ \langle\langle 35, 40\rangle, \langle 42, 47\rangle, \langle 49, 54\rangle\rangle, \\ \langle\langle 55, 60\rangle, \langle 62, 67\rangle, \langle 69, 74\rangle\rangle \end{array} \right\}.$$

In the remainder of this chapter, to simplify the notation and to allow for complications that will arise, we will use the following equivalent notation:

CTRY	CODE	CAP
$\langle 15, 20\rangle$	$\langle 22, 27\rangle$	$\langle 29, 34\rangle$
$\langle 35, 40\rangle$	$\langle 42, 47\rangle$	$\langle 49, 54\rangle$
$\langle 55, 60\rangle$	$\langle 62, 67\rangle$	$\langle 69, 74\rangle$

(6.1)

Before proceeding, let us note that even if one has perfect attribute recognizers, automatic wrapper construction is still important. The reason is that, while perfect, the recognizers may run slowly. However, a software agent needs its information extraction subsystem to run quickly; ideally, it should respond in real time to users' commands. Wrapper construction offers the prospect of off-line caching of the smart-but-slow recognizer's output in a form that can be used quickly on-line.

Imperfect recognizers. This example was so simple because each recognizer is perfect. Of course, we want to accomodate recognizers that make mistakes. There are two kinds of mistakes: *false positives* and *false negatives*. An instance returned by a recognizer is a false positive if it does not correspond to an actual value of the attribute. An actual value of the attribute is a false negative if it is not included among the instances returned by the recognizer. In general, an imperfect recognizer might either exhibit false positives, false negatives, or both. Thus with respect to mistakes, there are four possible kinds of recognizers:

perfect, neither false positives nor false negatives;
incomplete, false negatives but no false positives;
unsound, false positives but no false negatives; and
unreliable, both false positives and false negatives.

We now return to the example to see how these different kinds of recognizers effect the labeling problem.

Corroborating incomplete recognizers. Consider first incomplete recognizers, those that exhibit false negatives but not false positives. Suppose $\mathcal{R}_{\text{CTRY}}$ and $\mathcal{R}_{\text{CAP}}$ are incomplete while, as before, $\mathcal{R}_{\text{CODE}}$ is perfect. Suppose further that when given the example page, the recognizers respond as follows:

$$\begin{array}{|c|}\hline \mathcal{R}_{\text{CAP}} \\ \textit{incomplete} \\ \hline \langle 29, 34\rangle \\ \langle 69, 74\rangle \\ \hline \end{array} \quad \begin{array}{|c|}\hline \mathcal{R}_{\text{CTRY}} \\ \textit{incomplete} \\ \hline \langle 35, 40\rangle \\ \hline \end{array} \quad \begin{array}{|c|}\hline \mathcal{R}_{\text{CODE}} \\ \textit{perfect} \\ \hline \langle 22, 27\rangle \\ \langle 42, 47\rangle \\ \langle 62, 67\rangle \\ \hline \end{array} \tag{6.2}$$

Note that several of the instances that were originally reported are now missing. These missing instances are the false negatives that were incorrectly ignored.

The labeling problem is now more complicated. We must *corroborate* the output of the recognizers in an attempt to compensate for the imperfect information we receive about the label. This corroboration process produces the following label:

CTRY	CODE	CAP
?	$\langle 22, 27\rangle$	$\langle 29, 34\rangle$
$\langle 35, 40\rangle$	$\langle 42, 47\rangle$	?
?	$\langle 62, 67\rangle$	$\langle 69, 74\rangle$

(6.3)

We first explain this label, and then describe corroboration. Compare this label with the label listed at Equation 6.1, generated from three perfect recognizers. Note that the only difference is the "?" symbols. As before, each "cell" in the label indicates a single attribute value. The "?" symbol indicates that the label conveys no information about the instance in the given cell. For example, the second row of the label shows that the CTRY attribute occurs at position 35–40 and the CODE attribute occurs at position 42–47, but no information is known about the CAP attribute.

Earlier we saw that corroboration of perfect recognizers is very simple. Corroboration of incomplete recognizers is nearly as easy, so long as *at least one recognizer is perfect*. (We will formalize, motivate and exploit this assumption throughout this chapter.) The basic idea is that this perfectly-recognized attribute can be used to anchor the other instances.

Thus in the example, given the recognized instances, we know that (as before) the attribute ordering must be $\langle$CTRY, CODE, CAP$\rangle$. Therefore, by comparing the indices, it is a simple matter to "align" each CTRY and CAP instance with the correct CODE instance. For example, with which CODE instance does the CTRY instance $\langle 35, 40\rangle$ belong? To see why it is aligned with CODE instance $\langle 32, 37\rangle$, we need only note that: (1) $27 < 35$ and therefore CTRY $\langle 35, 40\rangle$ belongs to either the second or third CODE instances; and (2) $40 < 42$ and therefore CTRY $\langle 35, 40\rangle$ must belong to either the first or second CODE instances. Performing such analysis for each CTRY and CAP instance yields the label listed above.

These "pesudo-labels" that contain "?" symbols will be referred to as *partial labels*. Section 6.5 discusses how to extend the induction system to handle partially-labeled pages.

Multiple orderings. We have neglected an important complication: with imperfect recognizers, there might be more than one attribute ordering that is consistent with the recognized instances. For example, suppose that a library of two incomplete and one perfect recognizer output the following instances:

$\mathcal{R}_{\text{CAP}}$ *incomplete*
$\langle 49, 54\rangle$

$\mathcal{R}_{\text{CTRY}}$ *incomplete*
$\langle 35, 40\rangle$

$\mathcal{R}_{\text{CODE}}$ *perfect*
$\langle 22, 27\rangle$
$\langle 42, 47\rangle$
$\langle 62, 67\rangle$

In this situation, the original ordering $\langle \textsc{ctry}, \textsc{code}, \textsc{cap}\rangle$ as well as $\langle \textsc{cap}, \textsc{code}, \textsc{ctry}\rangle$ are both consistent with the recognized instances. To see this, note that each of the following labels would constitute a valid corroboration of the recognized instances:

CTRY	CODE	CAP
?	$\langle 22, 27\rangle$	?
$\langle 35, 40\rangle$	$\langle 42, 47\rangle$	$\langle 49, 54\rangle$
?	$\langle 62, 67\rangle$	?

CAP	CODE	CTRY
?	$\langle 22, 27\rangle$	$\langle 35, 40\rangle$
?	$\langle 42, 47\rangle$	?
$\langle 49, 54\rangle$	$\langle 62, 67\rangle$	?

Without additional information or constraints, the corroboration process has no basis for preferring one of these two labels over the other. On the other hand, the wrapper induction process that invokes the corroboration process might well be able to decide which is correct. Specifically, we'll see that it is extremely unlikely that there is a wrapper consistent with more than one of the orderings. Therefore, we will treat corroboration as a process that takes as input the recognizer library, and produces as output a *set* of partial labels, one for each consistent ordering. Corroboration returns all consistent partial labels; the induction system then tries to select the correct label (see Section 6.5 for details).

In this example, we want corroboration to compute the following set:

$$\left\{ \begin{array}{|c|c|c|} \hline \text{CTRY} & \text{CODE} & \text{CAP} \\ \hline ? & \langle 22, 27\rangle & ? \\ \langle 35, 40\rangle & \langle 42, 47\rangle & \langle 49, 54\rangle \\ ? & \langle 62, 67\rangle & ? \\ \hline \end{array} , \begin{array}{|c|c|c|} \hline \text{CAP} & \text{CODE} & \text{CTRY} \\ \hline ? & \langle 22, 27\rangle & \langle 35, 40\rangle \\ ? & \langle 42, 47\rangle & ? \\ \langle 49, 54\rangle & \langle 62, 67\rangle & ? \\ \hline \end{array} \right\}.$$

Corroborating unsound recognizers. The corroboration of incomplete recognizers required introducing the notion of partial labels and the "?" symbol, as well as machinery for handling multiple consistent attribute orderings. Unsound recognizers—those that exhibit false positives but not false negatives—involve additional modifications to the basic corroboration process.

Suppose $\mathcal{R}_{\text{CTRY}}$ is unsound while $\mathcal{R}_{\text{CODE}}$ and $\mathcal{R}_{\text{CAP}}$ are perfect. Suppose further that when given the example page, the recognizers respond as follows:

$\mathcal{R}_{\text{CAP}}$
perfect
$\langle 29, 34\rangle$
$\langle 49, 54\rangle$
$\langle 69, 74\rangle$

$\mathcal{R}_{\text{CTRY}}$
unsound
$\langle 5, 10\rangle$
$\langle 15, 20\rangle$
$\langle 21, 25\rangle$
$\langle 35, 40\rangle$
$\langle 55, 60\rangle$

$\mathcal{R}_{\text{CODE}}$
perfect
$\langle 22, 27\rangle$
$\langle 42, 47\rangle$
$\langle 62, 67\rangle$

Note that the CAP and CODE instances are unchanged from the first (perfect) example, while there are two additional CTRY instances, $\langle 5, 10\rangle$ and $\langle 21, 25\rangle$.

Corroboration proceeds as follows. As before, there is just one consistent ordering, $\langle$CTRY, CODE, CAP$\rangle$. Given this constraint, we can immediately insert the perfectly-recognized CODE and CAP instances into the output label. We can thus write down the following portion of the label:

CODE	CAP
$\langle 22, 27\rangle$	$\langle 29, 34\rangle$
$\langle 42, 47\rangle$	$\langle 49, 54\rangle$
$\langle 62, 67\rangle$	$\langle 69, 74\rangle$

We now discuss what to fill in for the CTRY column.

How are the remaining five CTRY instances handled? As before, we can conclude that the fourth and fifth instances, $\langle 35, 40\rangle$ and $\langle 55, 60\rangle$, belong to the second and third (respectively) CODE and CAP instances:

CTRY	CODE	CAP
	$\langle 22, 27\rangle$	$\langle 29, 34\rangle$
$\langle 35, 40\rangle$	$\langle 42, 47\rangle$	$\langle 49, 54\rangle$
$\langle 55, 60\rangle$	$\langle 62, 67\rangle$	$\langle 69, 74\rangle$

All that remains is to fill in the top-left cell of the label. The remaining three CTRY instances—$\langle 5, 10\rangle$, $\langle 15, 20\rangle$ and $\langle 21, 25\rangle$—are all (and the only) candidates for this cell. We know that exactly one of them is correct, but we don't yet know which. We can make some progress by discarding $\langle 21, 25\rangle$: this instance overlaps with the CODE instance $\langle 22, 27\rangle$; since $\mathcal{R}_{\text{CODE}}$ is perfect, we know that the CODE instances are correct, so $\langle 21, 25\rangle$ defers to $\langle 22, 27\rangle$.

Unfortunately, there is a no way for corroboration to decide between these two choices. We handle this situation by splitting the label into two, one for each possibility:

$$\left\{ \begin{array}{|c|c|c|} \hline \text{CTRY} & \text{CODE} & \text{CAP} \\ \hline \langle 5, 10\rangle & \langle 22, 27\rangle & \langle 29, 34\rangle \\ \langle 35, 40\rangle & \langle 42, 47\rangle & \langle 49, 54\rangle \\ \langle 55, 60\rangle & \langle 62, 67\rangle & \langle 69, 74\rangle \\ \hline \end{array}, \quad \begin{array}{|c|c|c|} \hline \text{CTRY} & \text{CODE} & \text{CAP} \\ \hline \langle 15, 20\rangle & \langle 22, 27\rangle & \langle 29, 34\rangle \\ \langle 35, 40\rangle & \langle 42, 47\rangle & \langle 49, 54\rangle \\ \langle 55, 60\rangle & \langle 62, 67\rangle & \langle 69, 74\rangle \\ \hline \end{array} \right\}$$

Note that these two labels are identical except for the contents of the top-left cell.

To summarize, some of the instances returned by an unsound recognizer might be false positives. In general, the corroboration system has no basis on which to decide which are false and which are true positives. Therefore, corroboration produces one label for each way to make a choice about which of the instances are false positives and which are true positives. In this light, corroboration amounts to guaranteeing that one of the returned labels corresponds to the correct choice.

Recall that in the earlier discussion of multiple consistent orderings, we mentioned that the induction system is responsible for deciding which ordering is correct. Similarly, the result of corroborating unsound recognizers is a set of labels; the induction system is responsible for deciding which is correct. We discuss these details in Section 6.5.

Corroborating perfect, unsound, and incomplete recognizers. We have seen examples of corroborating perfect and unsound recognizers, and perfect and incomplete recognizers. We now discuss corroboration when the recognizer library contains recognizers of all three types. As might be expected, in the general case there might be:

- multiple consistent attribute orderings, due to false negatives;
- partial labels, due to false negatives; and
- multiple labels for any given attribute ordering, due to ambiguous false positives.

For example, suppose that $\mathcal{R}_{\text{CODE}}$ is perfect, $\mathcal{R}_{\text{CTRY}}$ is incomplete, and $\mathcal{R}_{\text{CAP}}$ is unsound, and that these recognizers output the following instances:

$\mathcal{R}_{\text{CAP}}$ *unsound*
$\langle 5, 10\rangle$
$\langle 29, 34\rangle$
$\langle 40, 41\rangle$
$\langle 49, 54\rangle$
$\langle 55, 60\rangle$
$\langle 69, 74\rangle$

$\mathcal{R}_{\text{CTRY}}$ *incomplete*
$\langle 35, 40\rangle$

$\mathcal{R}_{\text{CODE}}$ *perfect*
$\langle 22, 27\rangle$
$\langle 42, 47\rangle$
$\langle 62, 67\rangle$

First consider the CTRY and CODE—the attribute that are recognized by perfect and incomplete recognizers. These following two labels are consistent with these instances:

$$\left\{ \begin{array}{|c|c|} \hline \text{CTRY} & \text{CODE} \\ \hline ? & \langle 22, 27\rangle \\ \langle 35, 40\rangle & \langle 42, 47\rangle \\ ? & \langle 62, 67\rangle \\ \hline \end{array} , \begin{array}{|c|c|} \hline \text{CODE} & \text{CTRY} \\ \hline \langle 22, 27\rangle & \langle 35, 40\rangle \\ \langle 42, 47\rangle & ? \\ \langle 62, 67\rangle & ? \\ \hline \end{array} \right\} . \tag{6.4}$$

Now consider the CAP attribute. When we combine the CAP instances with the first label, we see that CAP must occur as the third column; in this position, there are two subsets of the CAP instances that are consistent with the rest of the label:

CTRY	CODE	CAP
?	$\langle 22, 27\rangle$	$\langle 29, 34\rangle$
$\langle 35, 40\rangle$	$\langle 42, 47\rangle$	$\langle 49, 54\rangle$
?	$\langle 62, 67\rangle$	$\langle 69, 74\rangle$

CTRY	CODE	CAP
?	$\langle 22, 27\rangle$	$\langle 29, 34\rangle$
$\langle 35, 40\rangle$	$\langle 42, 47\rangle$	$\langle 55, 60\rangle$
?	$\langle 62, 67\rangle$	$\langle 69, 74\rangle$

Note that these labels are the same except for the middle-right cell. Consider the first label. Note that all of the other CAP instances—$\langle 29, 34\rangle$, $\langle 49, 54\rangle$ and $\langle 69, 74\rangle$—either overlap with instances already installed in the label (which must be true positives since they were reported by incomplete or perfect recognizers) or are inconsistent with CAP being the third attribute. The second label is derived by similar reasoning.

Next, we expand the second label in Equation 6.4, which produces the following two labels:

CAP	CODE	CTRY
$\langle 5,10\rangle$	$\langle 22,27\rangle$	$\langle 35,40\rangle$
$\langle 40,41\rangle$	$\langle 42,47\rangle$	?
$\langle 49,54\rangle$	$\langle 62,67\rangle$	?

CAP	CODE	CTRY
$\langle 5,10\rangle$	$\langle 22,27\rangle$	$\langle 35,40\rangle$
$\langle 40,41\rangle$	$\langle 42,47\rangle$	?
$\langle 55,60\rangle$	$\langle 62,67\rangle$	?

Thus the final result of corroboration is the following set of four partial labels:

$$\left\{ \begin{array}{cc} \begin{array}{|c|c|c|} \hline \text{CTRY} & \text{CODE} & \text{CAP} \\ \hline ? & \langle 22,27\rangle & \langle 29,34\rangle \\ \langle 35,40\rangle & \langle 42,47\rangle & \langle 49,54\rangle \\ ? & \langle 62,67\rangle & \langle 69,74\rangle \\ \hline \end{array}, & \begin{array}{|c|c|c|} \hline \text{CTRY} & \text{CODE} & \text{CAP} \\ \hline ? & \langle 22,27\rangle & \langle 29,34\rangle \\ \langle 35,40\rangle & \langle 42,47\rangle & \langle 55,60\rangle \\ ? & \langle 62,67\rangle & \langle 69,74\rangle \\ \hline \end{array}, \\ \\ \begin{array}{|c|c|c|} \hline \text{CAP} & \text{CODE} & \text{CTRY} \\ \hline \langle 5,10\rangle & \langle 22,27\rangle & \langle 35,40\rangle \\ \langle 40,41\rangle & \langle 42,47\rangle & ? \\ \langle 49,54\rangle & \langle 62,67\rangle & ? \\ \hline \end{array}, & \begin{array}{|c|c|c|} \hline \text{CAP} & \text{CODE} & \text{CTRY} \\ \hline \langle 5,10\rangle & \langle 22,27\rangle & \langle 35,40\rangle \\ \langle 40,41\rangle & \langle 42,47\rangle & ? \\ \langle 55,60\rangle & \langle 62,67\rangle & ? \\ \hline \end{array} \end{array} \right\}.$$

Unreliable recognizers. So far, we have ignored unreliable recognizers—those that exhibit both false positives and false negatives. The corroboration process is similar to what we have seen so far. However, recall that the induction system must iterate over all partial labels consistent with the recognized instances (see Section 6.5 for details). An important consideration, therefore, is the number of such labels produced by the corroboration process.

Unfortunately, we have observed in practice that this number becomes very large with unreliable recognizers, even when the rate at which mistakes are made is small. Therefore, in this chapter we largely ignore unreliable recognizers.

6.3 A formal model of corroboration

In the preceding section, we describe the major issues that must be tackled when using noisy attribute recognizers to solve the labeling problem. In this section we provide a formal model of these ideas and intuitions.

- We assume a set of *attribute identifiers* $\{F_1, \ldots, F_K\}$, where there are K attributes per tuple. In the example discussed in the previous section, $K = 3$ and the attribute identifiers were $\{\textsc{ctry}, \textsc{code}, \textsc{cap}\}$.

- An *instance* is a pair of non-negative integers $\langle b, e\rangle$ such that $b \leq e$. When not clear from context, the alternative notation $\langle b, e, F_k\rangle$ emphasizes that $\langle b, e\rangle$ is an instance of attribute F_k.

- A *recognizer* $\mathcal{R}$ is a function from a page to a set of instances:

  $$\mathcal{R} : \Sigma^* \rightarrow 2^{[1,\infty]\times[1,\infty]},$$

  where Σ^* is the set of all strings (see Appendix C). The notation $\mathcal{R}(P) = \{\ldots, \langle b, e\rangle, \ldots\}$ indicates that the result of applying recognizer $\mathcal{R}$ to page P is the indicated set of instances.

- The *recognizer library* is a set of recognizers $\Delta = \{\mathcal{R}_{F_1}, \ldots, \mathcal{R}_{F_K}\}$, exactly one per attribute.

- For a given page P and attribute F_k, let $R^*_{F_k}(P)$ be the set of *true instances* of F_k in P. To be sure, neither the induction system nor the recognizers have access to the true instances. However, we introduce this notion to make precise the kinds of mistakes a recognizer makes. Note that in this chapter we are concerned only with pages that are formatted according to a tabular layout. Thus for example, we have that $|\mathcal{R}^*_{F_1}(P)| = |\mathcal{R}^*_{F_2}(P)| = \cdots = |\mathcal{R}^*_{F_K}(P)|$; for details see Footnote 1 on page 24.

- Recognizer $\mathcal{R}_{F_k}$ is *perfect* if, for every page P, $\mathcal{R}^*_{F_k}(P) = \mathcal{R}_{F_k}(P)$. We abbreviate the assertion that $\mathcal{R}_{F_k}$ is perfect as $\textsc{perf}(\mathcal{R}_{F_k})$.

- Recognizer $\mathcal{R}_{F_k}$ is *incomplete* if, for every page P, $\mathcal{R}_{F_k}(P) \subseteq \mathcal{R}^*_{F_k}(P)$. The notation $\textsc{incom}(\mathcal{R}_{F_k})$ means that $\mathcal{R}_{F_k}$ is incomplete.

- Recognizer $\mathcal{R}_{F_k}$ is *unsound* if, for every page P, $\mathcal{R}^*_{F_k}(P) \subseteq \mathcal{R}_{F_k}(P)$. The notation $\textsc{unsnd}(\mathcal{R}_{F_k})$ means that $\mathcal{R}_{F_k}$ is unsound.

- Recognizer $\mathcal{R}_{F_k}$ is *unreliable* if it is not perfect, incomplete or unsound. As mentioned earlier, unreliable recognizers make corroboration more complicated, and so we largely ignore them in this chapter.

- An *attribute ordering* "$\prec$" is a total order over a set of attribute identifiers.

- A *label* is an array with K columns and M rows, with each *cell* in the label containing an instance. The headings in the ruled-table notation—*e.g.*, see Equation 6.1—emphasize the ordering over the attributes—*e.g.*, $\langle \textsc{ctry}, \textsc{code}, \textsc{cap} \rangle$ in Equation 6.1. Note that we use this tabular notation rather than the set notation introduced in Equation 2.3 (Chapter 2) just for convenience; the two notations are equivalent.

- The *true label* for a given page P is simply the result of appending the true instances for each attribute together into a label, with the attributes selected according to the unique valid ordering. For instance, in the following example, the three sets of true instances on the left give rise to the true label on the right:

$$\begin{array}{|c|}\hline \mathcal{R}^*_{\text{CODE}} \\ \hline \langle 22,27\rangle \\ \langle 42,47\rangle \\ \langle 62,67\rangle \\ \hline \end{array}\,,\;\begin{array}{|c|}\hline \mathcal{R}^*_{\text{CTRY}} \\ \hline \langle 15,20\rangle \\ \langle 35,40\rangle \\ \langle 55,60\rangle \\ \hline \end{array}\,,\;\begin{array}{|c|}\hline \mathcal{R}^*_{\text{CAP}} \\ \hline \langle 29,34\rangle \\ \langle 49,54\rangle \\ \langle 69,74\rangle \\ \hline \end{array}\quad\Rightarrow\quad\begin{array}{|c|c|c|}\hline \textsc{ctry} & \textsc{code} & \textsc{cap} \\ \hline \langle 15,20\rangle & \langle 22,27\rangle & \langle 29,34\rangle \\ \langle 35,40\rangle & \langle 42,47\rangle & \langle 49,54\rangle \\ \langle 55,60\rangle & \langle 62,67\rangle & \langle 69,74\rangle \\ \hline \end{array}$$

- A *partial label* is a label, except that each cell contains either an instance or the special symbol "?".

- The *labeling problem* is the following: given as input a page P and a library of recognizers $\{\mathcal{R}_{F_1}, \ldots, \mathcal{R}_{F_K}\}$, output a set $\{L_1, L_2, \ldots\}$ of possibly-partial labels. The notation $\langle P, \Delta \rangle$ indicates a particular labeling problem.

- Let L^* be the true label for page P. A set of partial labels is a *solution* to a given labeling problem if there exists an $L_j \in \{L_1, L_2, \ldots\}$ such that L_j *matches* L^*.

  Informally, partial label L_j matches true label L^* if L_j is identical to L^* for attributes recognized perfectly or unsoundly, but "holes" (indicated by "?") can occur in the incompletely-recognized columns, so long as none of the incompletely-recognized instances are discarded. For example, consider the labeling problem described by Equation 6.2, and assume that the true label is as shown in Equation 6.1. The partial label shown in Equation 6.3 matches the true label, because they are identical except for the three missing instances, and none of the "?" symbols were introduced spuriously.

  More precisely, partial label L_j matches true label L^* iff: (1) the attributes in L_j are ordered the same way as the attributes in L^*, and (2) L_j and L^* are identical except that a cell in column F_k can be the special symbol "?" iff $\textsc{incom}(\mathcal{R}_{F_k})$; and (3) for each F_k such that $\textsc{incom}(\mathcal{R}_{F_k})$, the set of non-"?" instances in column F_k must equal $\mathcal{R}_{F_k}(P)$.

6.4 The Corrob *algorithm*

In the previous section, we formally specified the labeling problem. In this section, we describe Corrob, an algorithm that correctly solves the labeling problem for the following special case:

> **Assumption 6.1 (Reasonable recognizers)** *The recognizer library Δ contains at least one perfect recognizer, and no unreliable recognizers.*

(We defer until Section 6.7 further discussion and justification of Assumption 6.1.)

Figure 6.1 lists the Corrob algorithm. In the remainder of this section, we first describe the algorithm in more detail by walking through an example, and then describing its formal properties.

6.4.1 Explanation of Corrob

As motivated in Section 6.2, the Corrob corroboration algorithm involves exploring a combinatorial space of all possible ways to combine the evidence from the recognizer library.

Perfect and incomplete recognizers are straightforward: since we allow partial labels, and since such recognizers never include false positives, we can simply incorporate the instances provided by these recognizers directly into every label output by Corrob. One important wrinkle is that incomplete recognizers give rise to an ambiguity over the ordering of attributes with the label, and so each possible ordering must be considered.

Unsound recognizers are somewhat more complicated: since any particular instance they report might be a false positive, we must in general allow for throwing away an arbitrary portion of the recognized instances. However, since unsound recognizers never exhibit false negatives, we know that the instances recognized by an unsound recognizer must contain all the true positives. Therefore, we must consider all ways of selecting exactly one recognized instance for each tuple, for each attribute recognized by an unsound recognizer.

The example. To describe the Corrob algorithm, we show its operation on the last example in Section 6.2. To repeat, there are three attributes, and the recognizer library $\Delta = \{\mathcal{R}_{\text{CTRY}}, \mathcal{R}_{\text{CAP}}, \mathcal{R}_{\text{CODE}}\}$ responds to an example page P as follows:

$\mathcal{R}_{\text{CAP}}$ *unsound*
$\langle 5, 10\rangle$
$\langle 29, 34\rangle$
$\langle 40, 41\rangle$
$\langle 49, 54\rangle$
$\langle 55, 60\rangle$
$\langle 69, 74\rangle$

$\mathcal{R}_{\text{CTRY}}$ *incomplete*
$\langle 35, 40\rangle$

$\mathcal{R}_{\text{CODE}}$ *perfect*
$\langle 22, 27\rangle$
$\langle 42, 47\rangle$
$\langle 62, 67\rangle$

```
$\underline{\textsf{Corrob}}$(page $P$, recognizer library $\Delta = \{\mathcal{R}_{F_1}, \ldots, \mathcal{R}_{F_K}\}$)
    $\mathcal{L} \leftarrow \{\}$
    $\textsf{NTPSet} \leftarrow \textsf{NTPSet}(P, \Delta)$                                        6.1(a)
    for each $\textsf{PTPSet} \in \textsf{PTPSets}(P, \Delta)$                                  6.1(b)
        for each "$\prec$" $\in \textsf{Orders}(\{F_1, \ldots, F_K\})$                          6.1(c)
            if $\textsf{Consistent?}(\text{“}\prec\text{”}, \textsf{NTPSet} \cup \textsf{PTPSet})$ then          6.1(d)
                $\mathcal{L} \leftarrow \mathcal{L} + \textsf{MakeLabel}(\text{“}\prec\text{”}, \textsf{NTPSet} \cup \textsf{PTPSet})$   6.1(e)
    return $\mathcal{L}$

$\underline{\textsf{NTPSet}}$(page $P$, recognizer library $\Delta$)
    Return the annotated instances from the perfect or incomplete recognizers:
            $\bigcup_{\mathcal{R}_{F_k}: \neg\textsc{unsnd}(\mathcal{R}_{F_k})} \textsf{Invoke}(P, \mathcal{R}_{F_k}(P))$

$\underline{\textsf{PTPSets}}$(page $P$, recognizer library $\Delta$)
    Let $M$ be the number of tuples in $P$'s true label:
        $M = |\mathcal{R}_{F_k}(P)|$ for some $\mathcal{R}_{F_k} \in \Delta$ such that $\textsc{perf}(\mathcal{R}_{F_k})$.
    Let $\Gamma(s, n)$ be the set of all subsets of set $s$ having of size $n$:
        $\Gamma(s, n) = \{s' \in 2^s \mid |s'| = n\}$.
    Return the set of ways to choose $M$ instances from each unsound recognizer:
            $\prod_{\mathcal{R}_{F_k}: \textsc{unsnd}(\mathcal{R}_{F_k})} \Gamma(\textsf{Invoke}(P, \mathcal{R}_{F_k}), M)$

$\underline{\textsf{Invoke}}$(page $P$, recognizer $\mathcal{R}_{F_k}$)
    Return the set of recognized instances, annotated with the attribute identifer:
            $\left\{\langle b, e, F_k\rangle \mid \langle b, e\rangle \in \mathcal{R}_{F_k}(P)\right\}$

$\underline{\textsf{Orders}}$(attribute identifiers $\{F_1, \ldots, F_K\}$)
    Return the set of all total orders "$\prec$" over the set $\{F_1, \ldots, F_K\}$

$\underline{\textsf{Consistent?}}$(ordering "$\prec$", annotated instances $\{\ldots, \langle b, e, F_k\rangle, \ldots\}$)
    Return $\textsc{true}$ iff there exists some label with attribute ordering "$\prec$"
    that contains every instances $\langle b, e, F_k\rangle$.

$\underline{\textsf{MakeLabel}}$(ordering "$\prec$", annotated instances $\{\ldots, \langle b, e, F_k\rangle, \ldots\}$)
    Assuming that $\textsf{Consistent?}(\text{“}\prec\text{”}, \{\ldots, \langle b, e, F_k\rangle, \ldots\}) = \textsc{true}$, return
    the label to which these inputs correspond.
```

Figure 6.1: *The* Corrob *algorithm.*

We now step through the execution of $\mathsf{Corrob}(P, \Delta)$.

Line 6.1(a): Necessarily true-positive instances. Corrob starts by invoking the perfect and incomplete recognizers—$\mathcal{R}_{\mathrm{CTRY}}$ and $\mathcal{R}_{\mathrm{CODE}}$—on page P. The NTPSet subroutine computes this set of *necessarily true positive* (NTP) instances. (The instances output by $\mathcal{R}_{\mathrm{CTRY}}$ and $\mathcal{R}_{\mathrm{CODE}}$—or more generally, all instances except those recognized unsoundly—are necessarily true positive instances, because incomplete and perfect recognizers never produce false positives.)

For book-keeping purposes that will become important at lines 6.1(d–e), we must keep track of the attribute to which each instance belongs. In the example, line 6.1(a) computes the following set of NTP instances:

$$\mathsf{NTPSet} \leftarrow \{\langle 35, 40, \textsc{ctry}\rangle,\ \langle 22, 27, \textsc{code}\rangle,\ \langle 42, 47, \textsc{code}\rangle,\ \langle 62, 67, \textsc{code}\rangle\}\,.$$

Line 6.1(b): Possibly true-positive instances. Corrob next collects and reasons about the instances recognized by the unsound recognizers—in this case, $\mathcal{R}_{\mathrm{CAP}}$. We know that some subset of the instances $\mathcal{R}_{\mathrm{CAP}}(P)$ are true positives and the remainder are false positives. Moreover, we know that there must be exactly three true positives, and therefore remaining $|\mathcal{R}_{\mathrm{CAP}}(P)| - 3$ instances are false positives. More generally, for each unsound recognizer, we know that M of the recognized instances are true positives and the remainder are false positives, where M is the number of instances in P's true label (note that we can determine M by consulting one of the perfect recognizers).

However, as discussed in Section 6.2, Corrob does not know which are correct and which are incorrect, and so it must try each way to select M choices from each unsound recognizer's set of instances. In this example, we must consider each of the following subsets:

$$\left\{\begin{array}{ccc}
\{\langle 5,10\rangle,\langle 29,34\rangle,\langle 40,41\rangle\}, & \{\langle 5,10\rangle,\langle 29,34\rangle,\langle 49,54\rangle\}, & \{\langle 5,10\rangle,\langle 29,34\rangle,\langle 55,60\rangle\}, \\
\{\langle 5,10\rangle,\langle 29,34\rangle,\langle 69,74\rangle\}, & \{\langle 5,10\rangle,\langle 40,41\rangle,\langle 49,54\rangle\}, & \{\langle 5,10\rangle,\langle 40,41\rangle,\langle 55,60\rangle\}, \\
\{\langle 5,10\rangle,\langle 40,41\rangle,\langle 69,74\rangle\}, & \{\langle 5,10\rangle,\langle 49,54\rangle,\langle 55,60\rangle\}, & \{\langle 5,10\rangle,\langle 49,54\rangle,\langle 69,74\rangle\}, \\
\{\langle 5,10\rangle,\langle 55,60\rangle,\langle 69,74\rangle\}, & \{\langle 29,34\rangle,\langle 40,41\rangle,\langle 49,54\rangle\}, & \{\langle 29,34\rangle,\langle 40,41\rangle,\langle 55,60\rangle\}, \\
\{\langle 29,34\rangle,\langle 40,41\rangle,\langle 69,74\rangle\}, & \{\langle 29,34\rangle,\langle 49,54\rangle,\langle 55,60\rangle\}, & \{\langle 29,34\rangle,\langle 49,54\rangle,\langle 69,74\rangle\}, \\
\{\langle 29,34\rangle,\langle 55,60\rangle,\langle 69,74\rangle\}, & \{\langle 40,41\rangle,\langle 49,54\rangle,\langle 55,60\rangle\}, & \{\langle 40,41\rangle,\langle 49,54\rangle,\langle 69,74\rangle\}, \\
\{\langle 40,41\rangle,\langle 55,60\rangle,\langle 69,74\rangle\}, & \{\langle 49,54\rangle,\langle 55,60\rangle,\langle 69,74\rangle\} &
\end{array}\right\}.$$

Notice that we have listed here the $\frac{6!}{3!\cdot(6-3)!} = 20$ ways to select $M = 3$ elements from the $|\mathcal{R}_{\mathrm{CAP}}(P)| = 6$ recognized CAP instances.

Each of these sets represents a collection of *possibly true-positive* (PTP) instances—*i.e.*, each set corresponds to a collection of instances that might be, though are not necessarily, true positives. The function PTPSets is invoked by line 6.1(b) to compute these PTP sets.

Specifically, PTPSets uses the function Γ to generate all ways to choose M elements from each unsound instance set, and then takes the cross product over all such sets-of-sets. In this example, PTPSets returns the above set of twenty sets (though for brevity we did not include the attribute annotations post-pended to each instance by the Invoke function).

DIGRESSION. This example is fairly simple, because there is only one unsound recognizer, and so the cross-product is trivial. However, in general there might be multiple unsound recognizers. For example, consider the following example involving two unsound recognizers, $\mathcal{R}_{F_1}$ and $\mathcal{R}_{F_2}$:

$\mathcal{R}_{F_1}$
unsound
$\langle 15,20\rangle$
$\langle 35,40\rangle$
$\langle 55,60\rangle$
$\langle 75,80\rangle$

$\mathcal{R}_{F_2}$
unsound
$\langle 25,30\rangle$
$\langle 45,50\rangle$
$\langle 65,70\rangle$
$\langle 85,90\rangle$

Suppose that there are $M = 2$ tuples in the true label. In this case we have that $\mathsf{PTPSets}$ returns the thirty-six member set

$$\left\{\begin{array}{ll} \{\langle 15,20,F_1\rangle,\langle 35,40,F_1\rangle\}, & \{\langle 15,20,F_1\rangle,\langle 55,60,F_1\rangle\}, \\ \{\langle 15,20,F_1\rangle,\langle 75,80,F_1\rangle\}, & \{\langle 35,40,F_1\rangle,\langle 55,60,F_1\rangle\}, \\ \{\langle 35,40,F_1\rangle,\langle 75,80,F_1\rangle\}, & \{\langle 55,60,F_1\rangle,\langle 75,80,F_1\rangle\} \end{array}\right\}$$
$$\times$$
$$\left\{\begin{array}{ll} \{\langle 25,30,F_2\rangle,\langle 45,50,F_2\rangle\}, & \{\langle 25,30,F_2\rangle,\langle 65,70,F_2\rangle\}, \\ \{\langle 25,30,F_2\rangle,\langle 85,90,F_2\rangle\}, & \{\langle 45,50,F_2\rangle,\langle 65,70,F_2\rangle\}, \\ \{\langle 45,50,F_2\rangle,\langle 85,90,F_2\rangle\}, & \{\langle 65,70,F_2\rangle,\langle 85,90,F_2\rangle\} \end{array}\right\}.$$

Note that each such PTP set contains two instances for F_1 and two instances for F_2, as required. END OF DIGRESSION.

Line 6.1(c): Attribute orderings. Corrob iterates over all choices for possibly true positive instances, as computed by PTPSets. Now, Corrob must also determine how the attributes are ordered. The function Orders returns the set of all $K!$ total orders over the attribute identifiers. In the example, we have:

$$\mathsf{Orders}(\{\textsc{ctry},\textsc{code},\textsc{cap}\}) = \left\{\begin{array}{ll} \langle\textsc{ctry},\textsc{code},\textsc{cap}\rangle, & \langle\textsc{ctry},\textsc{cap},\textsc{code}\rangle, \\ \langle\textsc{code},\textsc{ctry},\textsc{cap}\rangle, & \langle\textsc{code},\textsc{cap},\textsc{ctry}\rangle, \\ \langle\textsc{cap},\textsc{ctry},\textsc{code}\rangle, & \langle\textsc{cap},\textsc{code},\textsc{ctry}\rangle \end{array}\right\}.$$

In Figure 6.1, we use the notation "$\prec$" to refer to some element of this set. We write $F_k \prec F_{k'}$ to mean that F_k precedes $F_{k'}$ in the output of Orders to which "$\prec$" corresponds. For example, if "$\prec$" $= \langle\textsc{cap},\textsc{code},\textsc{ctry}\rangle$, then $\textsc{cap} \prec \textsc{ctry}$ because CAP precedes CTRY in $\langle\textsc{cap},\textsc{code},\textsc{ctry}\rangle$.

Line 6.1(d): Enumerating all combinations. As this point, the Corrob algorithm has gathered:

- the set NTPSet of necessarily true-positive instances;
- the set PTPSets, each member of which is a set of possible true-positive instances; and

- the set $\mathsf{Orders}(\{\textsc{ctry}, \textsc{code}, \textsc{cap}\})$ all possible attribute orderings.

The nested loop at lines 6.1(b-c) cause Corrob to iterate over the cross product of PTPSets and $\mathsf{Orders}(\{\textsc{ctry}, \textsc{code}, \textsc{cap}\})$. When coupled with NTPSet, each such combination of $\mathsf{PTPSet} \in \mathsf{PTPSets}$ and "$\prec$" $\in \mathsf{Orders}(\{\textsc{ctry}, \textsc{code}, \textsc{cap}\})$ corresponds to a complete guess for P's true label. In the example, there are $20 \times 6 = 120$ such combinations.

To verify each such guess, Corrob invokes the Consistent? function. Consistent? determines whether a given set of instances and a given attribute ordering can in fact be arranged in a tabular format. The basic idea is that an instance $\langle b, e, F_k\rangle$ conflicts with the remaining instances if it can not be inserted into a label with the given attribute ordering.

In the example, we must evaluate Consistent? against the 120 PTP/ordering pairs. For example, consider

$$\mathsf{Consistent?}\left(\underbrace{\langle \textsc{ctry}, \textsc{code}, \textsc{cap}\rangle,}_{1^{\underline{\mathrm{st}}}\text{ elt. of }\mathsf{Orders}} \underbrace{\left\{\begin{array}{c}\langle 35, 40, \textsc{ctry}\rangle,\\ \langle 22, 27, \textsc{code}\rangle,\\ \langle 42, 47, \textsc{code}\rangle,\\ \langle 62, 67, \textsc{code}\rangle\end{array}\right\}}_{\mathsf{NTPSet}} \cup \underbrace{\left\{\begin{array}{c}\langle 5, 10, \textsc{cap}\rangle,\\ \langle 29, 34, \textsc{cap}\rangle,\\ \langle 40, 41, \textsc{cap}\rangle\end{array}\right\}}_{1^{\underline{\mathrm{st}}}\text{ elt. of }\mathsf{PTPSets}}\right).$$

In this case Consistent? returns $\textsc{false}$. To see this, note that instance $\langle 5, 10, \textsc{cap}\rangle$ must occur in the same tuple as $\langle 22, 27, \textsc{code}\rangle$ (because there are no earlier $\textsc{code}$ instances), and therefore $\textsc{cap} \prec \textsc{code}$ because $\langle 5, 10\rangle$ precedes $\langle 22, 27\rangle$. However, the ordering $\langle \textsc{ctry}, \textsc{code}, \textsc{cap}\rangle$ requires that $\textsc{code} \prec \textsc{cap}$.

We will not illustrate Consistent? for the remaining 119 PTP/orderings combinations. Instead we simply assert only the following four pairs satisfy Consistent?:

$$\left\{\begin{array}{c}\langle\langle\textsc{ctry},\textsc{code},\textsc{cap}\rangle,\{\langle 29,34,\textsc{cap}\rangle,\langle 49,54,\textsc{cap}\rangle,\langle 69,74,\textsc{cap}\rangle\}\rangle,\\ \langle\langle\textsc{ctry},\textsc{code},\textsc{cap}\rangle,\{\langle 29,34,\textsc{cap}\rangle,\langle 55,60,\textsc{cap}\rangle,\langle 69,74,\textsc{cap}\rangle\}\rangle,\\ \langle\langle\textsc{cap},\textsc{code},\textsc{ctry}\rangle,\{\langle 5,10,\textsc{cap}\rangle,\langle 40,41,\textsc{cap}\rangle,\langle 49,54,\textsc{cap}\rangle\}\rangle,\\ \langle\langle\textsc{cap},\textsc{code},\textsc{ctry}\rangle,\{\langle 5,10,\textsc{cap}\rangle,\langle 40,41,\textsc{cap}\rangle,\langle 55,60,\textsc{cap}\rangle\}\rangle\end{array}\right\}.$$

Line 6.1(e): Building the labels. Finally, the MakeLabel function is invoked for every PTP/ordering combination that satisfies Consistent?. In the example, we invoke MakeLabel four times:

$$\mathsf{MakeLabel}\left(\langle\textsc{ctry},\textsc{code},\textsc{cap}\rangle,\left\{\begin{array}{cc}\langle 35,40,\textsc{ctry}\rangle, & \langle 22,27,\textsc{code}\rangle,\\ \langle 42,47,\textsc{code}\rangle, & \langle 62,67,\textsc{code}\rangle,\\ \langle 29,34,\textsc{cap}\rangle, & \langle 49,54,\textsc{cap}\rangle,\\ \langle 69,74,\textsc{cap}\rangle & \end{array}\right\}\right),$$

$$\mathsf{MakeLabel}\left(\langle\textsc{ctry},\textsc{code},\textsc{cap}\rangle,\left\{\begin{array}{cc}\langle 35,40,\textsc{ctry}\rangle, & \langle 22,27,\textsc{code}\rangle,\\ \langle 42,47,\textsc{code}\rangle, & \langle 62,67,\textsc{code}\rangle,\\ \langle 29,34,\textsc{cap}\rangle, & \langle 55,60,\textsc{cap}\rangle,\\ \langle 69,74,\textsc{cap}\rangle & \end{array}\right\}\right),$$

$$\mathsf{MakeLabel}\left(\langle\textsc{cap},\textsc{code},\textsc{ctry}\rangle,\left\{\begin{array}{cc}\langle 35,40,\textsc{ctry}\rangle, & \langle 22,27,\textsc{code}\rangle,\\ \langle 42,47,\textsc{code}\rangle, & \langle 62,67,\textsc{code}\rangle,\\ \langle 5,10,\textsc{cap}\rangle, & \langle 40,41,\textsc{cap}\rangle,\\ \langle 49,54,\textsc{cap}\rangle & \end{array}\right\}\right),$$

and

$$\mathsf{MakeLabel}\left(\langle\textsc{cap},\textsc{code},\textsc{ctry}\rangle,\left\{\begin{array}{cc}\langle 35,40,\textsc{ctry}\rangle, & \langle 22,27,\textsc{code}\rangle,\\ \langle 42,47,\textsc{code}\rangle, & \langle 62,67,\textsc{code}\rangle,\\ \langle 5,10,\textsc{cap}\rangle, & \langle 40,41,\textsc{cap}\rangle,\\ \langle 55,60,\textsc{cap}\rangle & \end{array}\right\}\right).$$

Each such invocation is straightforward: MakeLabel simply use the given attribute ordering and instances to construct an $M \times K$ array. Since MakeLabel is called only if the arguments satisfy Consistent?, we know that such an array is well defined. In the example, the invocations of MakeLabel result in the following four partial labels:

$$\left\{ \begin{array}{cc} \begin{array}{|c|c|c|} \hline \text{CTRY} & \text{CODE} & \text{CAP} \\ \hline ? & \langle 22,27 \rangle & \langle 29,34 \rangle \\ \langle 35,40 \rangle & \langle 42,47 \rangle & \langle 49,54 \rangle \\ ? & \langle 62,67 \rangle & \langle 69,74 \rangle \\ \hline \end{array}, & \begin{array}{|c|c|c|} \hline \text{CTRY} & \text{CODE} & \text{CAP} \\ \hline ? & \langle 22,27 \rangle & \langle 29,34 \rangle \\ \langle 35,40 \rangle & \langle 42,47 \rangle & \langle 55,60 \rangle \\ ? & \langle 62,67 \rangle & \langle 69,74 \rangle \\ \hline \end{array}, \\ \\ \begin{array}{|c|c|c|} \hline \text{CAP} & \text{CODE} & \text{CTRY} \\ \hline \langle 5,10 \rangle & \langle 22,27 \rangle & \langle 35,40 \rangle \\ \langle 40,41 \rangle & \langle 42,47 \rangle & ? \\ \langle 49,54 \rangle & \langle 62,67 \rangle & ? \\ \hline \end{array}, & \begin{array}{|c|c|c|} \hline \text{CAP} & \text{CODE} & \text{CTRY} \\ \hline \langle 5,10 \rangle & \langle 22,27 \rangle & \langle 35,40 \rangle \\ \langle 40,41 \rangle & \langle 42,47 \rangle & ? \\ \langle 55,60 \rangle & \langle 62,67 \rangle & ? \\ \hline \end{array} \end{array} \right\}.$$

This set is then returned by the Corrob algorithm as the output given inputs P and Δ.

6.4.2 Formal properties

In Section 6.3, we formally defined the labeling problem as well as what constitutes a solution. In Appendix B.7, we prove that Corrob always generates solutions:

> **Theorem 6.1 (Corrob is correct)** *For every recognizer library Δ and page P, if* Corrob$(P, \Delta) = \mathcal{L}$*, then $\mathcal{L}$ is a solution to the labeling problem $\langle P, \Delta \rangle$, provided that Assumption 6.1 holds of Δ.*

6.5 *Extending* Generalize$_{\text{HLRT}}$ *and the PAC model*

Ideally, the Induce inductive learning system would make use of the Corrob algorithm to label a set of example pages, and then provide these labeled pages to the generalization function Generalize$_{\text{HLRT}}$. Unfortunately, Generalize$_{\text{HLRT}}$ wants, and the PAC model assumes, perfectly labeled examples. In contrast:

- In general, Corrob produces multiple labels per example page, and guarantees only that one is correct.

- Moreover, incomplete recognizers mean that the resulting labels might be partial.

These differences mean that Corrob can not be used directly to label example pages for Generalize$_{\mathrm{HLRT}}$, and the PAC model of wrapper induction is not immediately applicable. Fortunately, only relatively minor extensions to the Generalize$_{\mathrm{HLRT}}$ algorithm and the PAC model are needed to accomodate Corrob's noisy labels. In this section we describe these enhancements.

To illustrate the basic idea, recall the country/code example in Figure 2.1. Suppose that instead of producing the correct label:

$$\begin{array}{l}
\texttt{<HTML><TITLE>Some Country Codes</TITLE><BODY>}\\
\texttt{<B>Some Country Codes</B><P>}\\
\texttt{<B>}\boxed{\texttt{Congo}}\texttt{</B> <I>}\boxed{\texttt{242}}\texttt{</I><BR>}\\
\texttt{<B>}\boxed{\texttt{Egypt}}\texttt{</B> <I>}\boxed{\texttt{20}}\texttt{</I><BR>}\\
\texttt{<B>}\boxed{\texttt{Belize}}\texttt{</B> <I>}\boxed{\texttt{501}}\texttt{</I><BR>}\\
\texttt{<B>}\boxed{\texttt{Spain}}\texttt{</B> <I>}\boxed{\texttt{34}}\texttt{</I><BR>}\\
\texttt{<HR><B>End</B></BODY></HTML>}
\end{array}$$

the corroboration algorithm Corrob produces the following incorrect label:

$$\begin{array}{l}
\texttt{<HTML><TITLE>Some Country Codes</TITLE><BODY>}\\
\texttt{<B>Some Country Codes</B><P>}\\
\texttt{<}\boxed{\texttt{B>Con}}\texttt{go</B> <I>}\boxed{\texttt{242}}\texttt{</I><BR>}\\
\texttt{<B>}\boxed{\texttt{Egypt}}\texttt{</B> <I>}\boxed{\texttt{20}}\texttt{</I><BR>}\\
\texttt{<B>}\boxed{\texttt{Belize}}\texttt{</B> <I>}\boxed{\texttt{501}}\texttt{</I><BR>}\\
\texttt{<B>}\boxed{\texttt{Spain</}}\texttt{B> <I>}\boxed{\texttt{34}}\texttt{</I><BR>}\\
\texttt{<HR><B>End</B></BODY></HTML>}
\end{array}$$

Note that two of the countries are labeled incorrectly. For example, this label might result from using an unsound country attribute recognizer.

What happens when we invoke Generalize$_{\mathrm{HLRT}}$ on this example? Generalize$_{\mathrm{HLRT}}$ certainly can not return some wrapper $w = \langle h, t, \ell_1, r_1, \ell_2, r_2 \rangle$, because no such wrapper exists. For instance, there is no valid value for ℓ_1, because the first country instance `B>Con` is preceded by the string "`<`" while the other three country instances are preceded by "`<B>`", and "`<`" and "`<B>`" share no common suffix. Moreover, note that due to the fourth country instance, `Spain</`, there is no acceptable value for r_1. In short, the given labeled page simply can not be wrapped by HLRT.

This example illustrates that even though Corrob alone can not decide which of its outputs labels are correct, a modified version of Generalize$_{\text{HLRT}}$ algorithm can decide, by checking to see whether a wrapper exists. If no wrapper exists, then the label must be incorrect, and Generalize$_{\text{HLRT}}$ can try the next label suggested by Corrob. Since Corrob is correct (Theorem 6.1), this simple strategy works, because eventually Generalize$_{\text{HLRT}}$ will encounter the correct label.

However, note that there is always some chance that this strategy will fail: it might be the case the some of the labels are incorrect, yet a consistent wrapper does indeed exists. This would cause the learning algorithm system to think it has succeeded when in fact it has learned the wrong wrapper. Later in this section we describe how to modify PAC model to account for the (ideally, very small) chance that a wrapper can be found that is consistent with an set of examples that are incorrectly labeled. Note that we have good reason to be optimistic: as the example illustrates, in the context of wrapper induction, mislabeling a page even slightly will very likely result in an unwrappable page.

So far, we have discussed the difficulties that result from unsound recognizers. Suppose that instead of being unsound, the country recognizer were incomplete. This might result in the following partial label:

```
<HTML><TITLE>Some Country Codes</TITLE><BODY>
<B>Some Country Codes</B><P>
<B>Congo</B> <I>242</I><BR>
<B>Egypt</B> <I>20</I><BR>
<B>Belize</B> <I>501</I><BR>
<B>Spain</B> <I>34</I><BR>
<HR><B>End</B></BODY></HTML>
```

In this case, so long as Generalize$_{\text{HLRT}}$ is somewhat careful in the way it processes this example, the induction algorithm can still produce the wrapper $w =$

$\langle$`<P>`, `<HR>`, `<B>`, `</B>`, `<I>`, `</I>`$\rangle$. The only problem is that the algorithm is generalizing on the basis of fewer "examples" of ℓ_1 and r_1—but this is essentially the same scenario as if the original example contained only two tuples instead of four.

This second example illustrates that partial labels—the "?" symbols in a partial label—are easily handled by $\mathsf{Generalize}_{\mathrm{HLRT}}$. Induction must simply rely on fewer examples of each HLRT component. Similarly, the PAC model must be extended so that the chance of learning each individual component is tailored on the basis of the evidence available about that specific component.

In the remainder of this section, we describe these extensions to $\mathsf{Generalize}_{\mathrm{HLRT}}$, and then discuss the modified PAC model.

6.5.1 The $\mathsf{Generalize}_{\mathrm{HLRT}}^{\mathrm{noisy}}$ *algorithm*

$\mathsf{Generalize}_{\mathrm{HLRT}}^{\mathrm{noisy}}$ is the name of our extension to $\mathsf{Generalize}_{\mathrm{HLRT}}$. While $\mathsf{Generalize}_{\mathrm{HLRT}}$ takes as input a set

$$\{\langle P_1, L_1\rangle, \ldots, \langle P_N, L_N\rangle\}$$

of perfectly-labeled examples, $\mathsf{Generalize}_{\mathrm{HLRT}}^{\mathrm{noisy}}$ takes as input a set

$$\left\{\langle P_1, \{L_1^1, L_1^2, \ldots\}\rangle, \ldots, \langle P_N, \{L_N^1, L_N^2, \ldots\}\rangle\right\}$$

of noisily-labeled examples. The intent is that the L_n^j are produced by Corrob: for each n,

$$\{L_n^1, L_n^2, \ldots\} = \mathsf{Corrob}(P_n, \Delta).$$

The algorithm. The $\mathsf{Generalize}_{\mathrm{HLRT}}^{\mathrm{noisy}}$ algorithm is shown in Figure 6.2. $\mathsf{Generalize}_{\mathrm{HLRT}}^{\mathrm{noisy}}$ operates by repeatedly invoking the original $\mathsf{Generalize}_{\mathrm{HLRT}}$ algorithm, once for each way to select one of the possible labels for each of the pages. The algorithm terminates when such a labeling yields a consistent wrapper.

```
$\underline{\mathsf{Generalize}^{\text{noisy}}_{\mathrm{HLRT}}}$(noisy examples $\mathcal{E} = \{\langle P_1, \{L_1^1, L_1^2, \ldots\}\rangle, \ldots, \langle P_N, \{L_N^1, L_N^2, \ldots\}\rangle\}$
    for each vector $\langle L_1^{j_1}, \ldots, L_N^{j_N}\rangle \in \{L_1^1, L_1^2, \ldots\} \times \cdots \times \{L_N^1, L_N^2, \ldots\}$        6.2(a)
        $w \leftarrow \mathsf{Generalize}_{\mathrm{HLRT}}(\{\langle P_1, L_1^{j_1}\rangle, \ldots, \langle P_N, L_N^{j_N}\rangle\})$
        if $w \neq$ FALSE then return $w$
```

Note that the Generalize$_{\mathrm{HLRT}}$ algorithm has been slightly modified from Figure 4.6; see the text and Figure 6.3 for details.

Figure 6.2: *The* Generalize$^{\text{noisy}}_{\mathrm{HLRT}}$ *algorithm, a modification of* Generalize$_{\mathrm{HLRT}}$ *(Figure 4.6) which can handle noisily labeled examples.*

Modifications to Generalize$_{\mathrm{HLRT}}$. As indicated in Figure 6.2, Generalize$^{\text{noisy}}_{\mathrm{HLRT}}$ relies on two minor modifications to the original Generalize$_{\mathrm{HLRT}}$ algorithm[1]—see Figure 6.3.

First, as described in Figure 4.6, Generalize$_{\mathrm{HLRT}}$ assumes that there exists a wrapper consistent with the input examples, and so the only termination condition explicitly specified involves finding such a wrapper. In contrast, Generalize$^{\text{noisy}}_{\mathrm{HLRT}}$ requires that Generalize$_{\mathrm{HLRT}}$ be modified so that it terminates and returns FALSE if no such consistent wrapper can be found. Implementing this change is very simple: we need only insert the three lines marked "†" in Figure 6.3. These lines ensure that the loops at lines 6.3(a), 6.3(d) and 6.3(g) terminated successfully. If any of these loops terminate without finding a legitimate value for the HLRT component, then Generalize$_{\mathrm{HLRT}}$ simply stops immediately and returns FALSE. Note that this modification to Generalize$_{\mathrm{HLRT}}$ has no bearing on the analysis and discussion of its behavior that we have presented.

[1] Recall that in Chapter 4, we actually defined two different HLRT generalization functions: the slower Generalize$_{\mathrm{HLRT}}$ algorithm (Figure 4.4), and the faster Generalize$^{*}_{\mathrm{HLRT}}$ algorithm (Figure 4.6). Then, since Generalize$^{*}_{\mathrm{HLRT}}$ dominates Generalize$_{\mathrm{HLRT}}$, we dropped the "$*$" annotation and used the symbol Generalize$_{\mathrm{HLRT}}$ to refer only to faster algorithm. The modifications to Generalize$_{\mathrm{HLRT}}$ proposed in this section apply in principle to either algorithm. However, we provide details only for the faster algorithm, since there is no point in worrying about the slower algorithm.

```
Generalize*_HLRT(examples $\mathcal{E} = \{\langle P_1, L_1\rangle, \ldots, \langle P_N, L_N\rangle\}$)
  for each $1 \leq k \leq K$                                                          6.3(a)
    for $r_k \leftarrow$ each prefix of some $P_n$'s intra-tuple separator for $S_{m,k}$           6.3(b)
      accept $r_k$ if C1 holds of $r_k$ and every $\langle P_n, L_n\rangle \in \mathcal{E}$          6.3(c)
    return FALSE if no candidate satisfies C1                                         †
  for each $1 < k \leq K$                                                          6.3(d)
    for $\ell_k \leftarrow$ each suffix of some $P_n$'s intra-tuple separator $S_{m,k-1}$          6.3(e)
      accept $\ell_k$ if C2 holds of $\ell_k$ and every $\langle P_n, L_n\rangle \in \mathcal{E}$          6.3(f)
    return FALSE if no candidate satisfies C2                                         †
  for $\ell_1 \leftarrow$ each suffix of some $P_n$'s head $S_{0,K}$                                6.3(g)
    for $h \leftarrow$ each substring of some $P_n$'s head $S_{0,K}$                              6.3(h)
      for $t \leftarrow$ each substring of some $P_n$'s tail $S_{M,K}$                             6.3(i)
        accept $\ell_1$, $h$, and $t$ if C3 holds of $\ell_1$, $h$, $t$ and every $\langle P_n, L_n\rangle \in \mathcal{E}$   6.3(j)
  return FALSE if no triplet of candidates satisfies C3                              †
  return $\langle h, t, \ell_1, r_1, \ldots, \ell_K, r_K\rangle$                                       6.3(k)
```

Figure 6.3: *A slightly modified version of the* Generalize$_{\text{HLRT}}$ *algorithm; compare with Figure 4.6.*

Second, we must modify Generalize$_{\text{HLRT}}$ so that it correctly processes labels containing "?" symbols. The discussion earlier gave the basic idea: rather than assuming that each label provides exactly one value for each attribute within each tuple, Generalize$_{\text{HLRT}}$ must reason only about those parts of the page about which the labels provides information.

Consider an example noisily-labeled page

```
<HTML><TITLE>Some Country Codes</TITLE><BODY>
<B>Some Country Codes</B><P>
<B>Congo</B> <I>242</I><BR>
<B>Egypt</B> <I>20</I><BR>
<B>Belize</B> <I>501</I><BR>
<B>Spain</B> <I>34</I><BR>
<HR><B>End</B></BODY></HTML>
```

This page's label contains two "?" symbols, one for each missing country. Now recall lines 4.6(a–c) of the Generalize$_{\text{HLRT}}$ algorithm (Figure 4.4). At this stage the algorithm is finding a value for each r_k, for $1 \leq k \leq K$. To do so the algorithm:

1. gathers a suitable set of candidates for r_k (namely, the prefixes of page P_1's intra-tuple separator for $S_{1,k}$); and then
2. tests each such candidate to see whether is satisfies predicate **C1**.

To extend Generalize$_{\text{HLRT}}$ so that it can handle partial labels, these two steps are modified as follows:

1. The candidate set for r_k contains the prefixes of *some* page P_n's intra-tuple separator for $S_{m,k}$. If (as is the case in the example above) page P_1's label is missing the first attribute of the first tuple, then algorithm simply uses *any other* page P_n instead of always using P_1, or *any other* tuple m instead of always the first tuple. Note that these choices do not matter—any such $S_{m,k}$ provides a satisfactory set of candidates for r_k. In the example, if we are trying to learn r_1 (*i.e.*, $k = 1$) then we can use as the candidates for r_1 all prefixes of the page's $S_{2,1}$ value `</B> <I>`, because $S_{1,1}$ is undefined (since $\langle b_{1,1}, e_{1,1}\rangle$ is missing from the page's label).
2. Recall predicate **C1** from Figure 4.3:

$$\mathbf{C1}(r_k, \langle P, L\rangle) \iff \forall_{1\leq m\leq M} \left(S_{m,k}/r_k = S_{m,k} \ \wedge \ F_{m,k}/r_k = \Diamond\right).$$

   Rather than checking *all* $1 \leq m \leq M$, we must check only those m such that instance $\langle b_{m,k}, e_{m,k}\rangle$ is present in the label. In the example, when learning r_1, the modified version of Generalize$_{\text{HLRT}}$ tests only $m = 2$ and $m = 4$; $m = 1$ and $m = 3$ are ignored because instances $\langle b_{1,1}, e_{1,1}\rangle$ and $\langle b_{3,1}, e_{3,1}\rangle$ are missing in the page's label.

This second modification to Generalize$_{\text{HLRT}}$ can be summarized as follows. As we've seen, Generalize$_{\text{HLRT}}$ assumes that labels contained no "?" symbols, and thus the algorithm was described in very simple terms. But we have shown how, for the learning of r_k, it is very simple to generalize the specification of the algorithm so that "?" symbols are handled properly. A similar set of changes are made for Generalize$_{\text{HLRT}}$'s other steps.

Summary. In Chapter 4 and this section, we have described several different versions of the HLRT generalization algorithm. As Footnote 1 suggests, these different versions can be somewhat confusing. Therefore, we summarize this discussion by listing the four algorithms:

$\mathsf{Generalize}_{\mathrm{HLRT}}$ **(Figure 4.4)**—the original HLRT generalization function. While conceptually elegant, this algorithm is extremely inefficient.

$\mathsf{Generalize}^{*}_{\mathrm{HLRT}}$ **(Figure 4.6)**—a re-implementation of $\mathsf{Generalize}_{\mathrm{HLRT}}$ that is substantially more efficient. Since $\mathsf{Generalize}^{*}_{\mathrm{HLRT}}$ so clearly dominates $\mathsf{Generalize}_{\mathrm{HLRT}}$, we largely ignore the straw-man algorithm, and use the symbol $\mathsf{Generalize}_{\mathrm{HLRT}}$ to refer to the improved version.

$\mathsf{Generalize}^{\mathrm{noisy}}_{\mathrm{HLRT}}$ **(Figure 6.2)**—the HLRT generalization function that handles the noisy labels generated by Corrob by calling (a modified version of) $\mathsf{Generalize}_{\mathrm{HLRT}}$ as a subroutine.

Modified $\mathsf{Generalize}_{\mathrm{HLRT}}$ **(Figure 6.3)**—two small changes to the $\mathsf{Generalize}_{\mathrm{HLRT}}$ algorithm ($\mathsf{Generalize}^{*}_{\mathrm{HLRT}}$ to be precise) that were just described. Since these modifications are so minor, we did not dignify them by introducing a new symbol.

6.5.2 Extending the PAC model

We have just described $\mathsf{Generalize}^{\mathrm{noisy}}_{\mathrm{HLRT}}$, which repeatedly invokes the $\mathsf{Generalize}_{\mathrm{HLRT}}$ algorithm described in Chapter 4, thereby enabling wrapper induction in the face of noisy example labels. We now show how to extend the PAC, developed in Section 4.6, to handle noisy labels.

PAC analysis provides an answer to a fundamental question in inductive learning: how many examples must a learner see before its output hypothesis is to be trusted? As discussed earlier, there are two main problems with directly applying the PAC model developed earlier, both of which derive from the fact that the model assumes correct labels.

First, the fact that the noisy labels might be *partial*—*i.e.*, include "?" symbols—means that we must be careful when calculating the chance that a wrapper has low error, since these "?" symbols effectively mask parts of the page from the learner's attempt to determine a wrapper. Without special care, the PAC model will overestimate the reliability of a learned wrapper.

The second problem is that unsound recognizers might return false-positive instances, which might then be incorrectly inserted into a page's label. What if, by chance, a wrapper consistent with these incorrectly labeled pages can be found? The result is an incorrect wrapper—for the examples (and possibly other pages as well) the wrapper extracts the *wrong* information content (namely, the text indicated by the incorrect labels). To deal with this situation, we must extend the PAC model so that it does not overestimate the reliability of a learned wrapper by ignoring the fact that the wrapper might have been learned on the basis of noisy labels.

Handling partial labels. The solution to the problem of partial labels is straightforward. We must do some additional bookkeeping to ensure that we count the actual number of examples of each delimiter. Recall the original PAC model (Equation 4.5):

$$\Psi(K)\left(1-\frac{\epsilon}{\Psi(K)}\right)^{M_{\mathrm{tot}}} \;+\; \Phi(R)\left(1-\frac{\epsilon}{2}\right)^{N} \;<\; \delta,$$

where the set of examples $\mathcal{E} = \{\ldots,\langle P_n, L_n\rangle,\ldots\}$ consists of $N = |\mathcal{E}|$ examples, $M_{\mathrm{tot}} = \sum_n M_n$ is the total number of tuples in $\mathcal{E}$, page P_n contains $M_n = |L_n|$ tuples, each tuple consists of K attributes, and the shortest example page has length $R = \min_n |P_n|$), and $\Psi(K)$ and $\Phi(R)$ are defined by Equations 4.6 and 4.7 on page 67.

In this original model, the parameter N indicates the number of observations from which the head (h), tail (t) and ℓ_1 HLRT components are learned. Similarly, M_{tot} counts the number of observations from which the remaining components (ℓ_2, …, ℓ_K, r_1, …, r_K) are learned. Accommodating partial labels is a matter of refining these counts to reflect the partial labels' incomplete information.

We first walk through an example, and then provide the details. Consider the following noisily-labeled page:

```
<HTML><TITLE>Some Country Codes</TITLE><BODY>
<B>Some Country Codes</B><P>
<B>Congo</B> <I>242</I><BR>
<B>Egypt</B> <I>20</I><BR>
<B>Belize</B> <I>501</I><BR>
<B>Spain</B> <I>34</I><BR>
<HR><B>End</B></BODY></HTML>
```

What can we learn from this example? The four intact code instances provide evidence for ℓ_2 and r_2, the left- and right-hand delimiters for the second attribute, the country code in this example. Specifically, for each of these two delimiters, we have four example strings from which to generalize.

However, note that we have less evidence about ℓ_1 and r_1, the left- and right-hand delimiters for the country attribute. Since just two of the instances are present, the example provides only two example strings from which to generalize ℓ_1 and r_1.

Finally, consider the head delimiter h and the tail delimiter t. Since the final (in this example, second) attribute of the last tuple is not missing, this example provides us with a complete example of a page's tail region, which provides evidence for determining t. In contrast, since the first attribute of the first tuple is missing, the example does not provide evidence for a page's head, and so we can learn nothing about the value of h.

We can summarize these observation as follows. This example illustrates that it is not correct to assume that the examples provide exactly N observations of page heads, N observations of page tails, and M_{tot} observations of each attribute.

Instead, we proceed as follows. In the new PAC analysis, we are interested in a set

$$\mathcal{E} = \left\{ \langle P_1, L_1^{j_1} \rangle, \ldots, \langle P_N, L_N^{j_N} \rangle \right\}$$

of examples, where each $L_n^{j_n}$ is a (possibly partial) label:

F_1	F_2		F_K
$\langle b_{1,1}, e_{1,1}\rangle$	$\langle b_{1,2}, e_{1,2}\rangle$		$\langle b_{1,K}, e_{1,K}\rangle$
$\vdots$	$\vdots$	$\cdots$	$\vdots$
$\langle b_{M_n,1}, e_{M_n,1}\rangle$	$\langle b_{M_n,2}, e_{M_n,2}\rangle$		$\langle b_{M_n,K}, e_{M_n,K}\rangle$

(Importantly, note that, though not shown explicitly, since each L_n^{jn} might be partial, some of the $\langle b_{m,k}, e_{m,k}\rangle$ might be missing, with the symbol "?" occurring instead.)

The extended PAC model is stated in terms of the following parameters of $\mathcal{E}$:

$$\begin{aligned}
M_n^k &= \text{the number of non-missing instances of attribute } k \text{ on page } P_n \\
M_{\text{tot}}^k &= \sum_n M_n^k \\
N_n^* &= \begin{cases} 0 & \text{if } \langle b_{1,1}, e_{1,1}\rangle \text{ or } \langle b_{M_n,K}, e_{M_n,K}\rangle \text{ is missing on page } P_n \\ 1 & \text{otherwise} \end{cases} \\
N^* &= \sum_n N_n^*
\end{aligned}$$

These new parameters are interpreted as follows:

- M_n^k is the number of instances of attribute F_k on page P_n. The K variables M_n^k thus generalize the role of the single variable M_n in the original model. If label L_n^{jn} is not partial, then all the M_n^k's are equal: $M_n^1 = \cdots = M_n^K = M_n$.
- The K values of M_{tot}^k generalize the role of M_{tot}. If none of the labels are partial, then $M_{\text{tot}}^1 = \cdots = M_{\text{tot}}^K = M_{\text{tot}}$.
- N_n^* indicates whether page P_n's head and tail can both be identified. With no partial labels, the head and tail can always be identified: $N_n^* = 1$.
- N^* generalize the role of N in the original model. With no partial labels, $N^* = N$.

In the country/code example mentioned earlier, if we call the example page P_1, then we have $M_1 = 4$ while $M_1^1 = 2$ and $M_1^2 = 4$, and $N_1^* = 0$. If the set of examples consists *only* of page P_1, then we have $M_{\text{tot}} = 4$ while $M_{\text{tot}}^1 = 2$ and $M_{\text{tot}}^2 = 4$, and $N = 1$ while $N^* = 0$.

These new parameters are then incorporated into the PAC model as follows. First, the first term in the left-hand side of Equation 4.5 changes as follows:

$$\Psi(K)\left(1-\frac{\epsilon}{\Psi(K)}\right)^{M_{\mathrm{tot}}} \quad \Longrightarrow \quad 2\left(\left(1-\frac{\epsilon}{\Psi(K)}\right)^{M_{\mathrm{tot}}^{1}} + 2\sum_{k=2}^{K}\left(1-\frac{\epsilon}{\Psi(K)}\right)^{M_{\mathrm{tot}}^{k}}\right)$$

(recall from Equation 4.6 that $\Psi(K) = 4K - 2$).

This change implements the more sophisticated bookkeeping that is needed to handle partial labels. Specifically, if we examine the proof of Theorem 4.8 (Section B.4), we find that Theorem 4.8 assumed that $M_n^1 = \cdots = M_n^K = M_n$ for each n; the described change generalizes the PAC model. Note that, as expected, if the recognizers are perfect, then $M_n^1 = \cdots = M_n^K = M_n$, and the change has no effect.

Next, the second term in the left-hand side of Equation 4.5 is changed as follows:

$$\Phi(R)\left(1-\frac{\epsilon}{2}\right)^{N} \quad \Longrightarrow \quad \Phi(R)\left(1-\frac{\epsilon}{2}\right)^{N^*}$$

Again, if the recognizers are perfect, then $N^* = N$, and this change has no effect.

Putting these two modifications together, we arrive at the following equation:

$$\left(\begin{array}{c} 2\left(\left(1-\frac{\epsilon}{\Psi(K)}\right)^{M_{\mathrm{tot}}^{1}} + 2\sum_{k=2}^{K}\left(1-\frac{\epsilon}{\Psi(K)}\right)^{M_{\mathrm{tot}}^{k}}\right) \\ + \\ \Phi(R)\left(1-\frac{\epsilon}{2}\right)^{N^*} \end{array}\right) < \delta \qquad (6.5)$$

To summarize, the new PAC model (Equation 6.5) generalizes the original model (Equation 4.5) so that the PAC analysis correctly handles partially labeled pages.

Handling false positives. Recall that, from a PAC-theoretic perspective, the difficulty with false positives is that there is no guarantee that there does not exist some wrapper that happens to be consistent with a set of incorrectly-labeled examples.

However, suppose that we have a bound μ on the chance that there exists a wrapper consistent with a set of incorrectly labeled examples. More formally, suppose that, for any set $\mathcal{E} = \{\langle P_1, L_1^{j_1}\rangle, \ldots, \langle P_N, L_N^{j_N}\rangle\}$ of examples, if $\mathcal{E}$ contains at least one incorrectly labeled example, then

$$\Pr\left[\exists_{w \in \mathcal{H}_{\mathrm{HLRT}}} \forall_{\langle P_n, L_n^{j_n}\rangle} w(P_n) = L_n^{j_n}\right] \leq \mu. \tag{6.6}$$

The bound μ provides the PAC model with some measure of how "dangerous" false positives are in a particular domain. In this Chapter, we have provided examples meant to motivate the claim that μ is relatively small in the HTML applications with which this thesis is concerned. In the Section 7.7, we return to this claim, and find that μ is extremely close to zero; for now, we simply assume that μ is provided as an additional parameter for the PAC model.[2]

Before proceeding, note that μ is *not* the chance that a label is wrong. As we have seen, $\mathsf{Generalize}_{\mathrm{HLRT}}^{\mathrm{noisy}}$ assumes that a consistent wrapper can be found only for correctly labeled pages; μ is a measure of how risky this assumption is. For example, our recognizers library Δ might make many mistakes, so that nearly all the labels returned by Corrob^* contain errors. Nevertheless, μ can be close to zero, if the information resource under consideration is sufficiently structured so that incorrect labels are usually discovered.

Incorporating μ into the model is straightforward. As discussed in Section 4.8, the terms on the left-hand side of Equations 4.5 and 6.5 account for different ways in which the induction process might fail to generate a high-quality wrapper. We simply must introduce μ as an additional term:

$$\left(\begin{array}{c} 2\left(\left(1 - \dfrac{\epsilon}{\Psi(K)}\right)^{M_{\mathrm{tot}}^{1}} + \ 2\displaystyle\sum_{k=2}^{K}\left(1 - \dfrac{\epsilon}{\Psi(K)}\right)^{M_{\mathrm{tot}}^{k}}\right) \\ + \\ \Phi(R)\left(1 - \frac{\epsilon}{2}\right)^{N^*} \\ + \\ \mu \end{array}\right) \quad < \quad \delta \tag{6.7}$$

[2] In Chapter 8, we describe how this model of noise is related to others in the PAC literature.

Note that if there is *no* chance that an incorrect label will yield a consistent wrapper, then $\mu = 0$ and the new PAC model (Equation 6.7) reduces to the previous model (Equation 6.5).

In summary, Equation 6.7 defines the generalized PAC model, which correctly accounts for both incomplete and unsound recognizers.

6.6 Complexity analysis and the Corrob* *algorithm*

The $\mathsf{Generalize}_{\mathrm{HLRT}}^{\mathrm{noisy}}$ algorithm described above is a relatively clean extension of the original $\mathsf{Generalize}_{\mathrm{HLRT}}$ algorithm so that it handles pages that are incorrectly labeled. However, a simple analysis reveals that it is computationally very expensive. In this section we point out these computational costs, and describe a modified version of Corrob, called Corrob*, which overcomes these costs.

The $\mathsf{Generalize}_{\mathrm{HLRT}}^{\mathrm{noisy}}$ takes as input a set

$$\left\{\langle P_1, \{L_1^1, \ldots, L_1^{J_1}\}\rangle, \ldots, \langle P_N, \{L_N^1, \ldots, L_N^{J_N}\}\rangle\right\}$$

of noisily labeled pages generated by the Corrob algorithm, and invokes the original $\mathsf{Generalize}_{\mathrm{HLRT}}$ algorithm once for each of the $\prod_n J_n$ ways to choose one of the candidate labels for each page, where (as indicated above) page Corrob returns J_n possible labels for page P_n.

How large can each J_n be? Examination of the Corrob algorithm (Figure 6.1) provides the following bounds. Line 6.1(c) indicates that each of the J_n labels for page P_n potentially corresponds to a different ordering of the K attributes, and there are $K!$ such orderings. Furthermore, suppose that there are U $(< K)$ unsound recognizers in the library Δ provided to Corrob. Suppose further that each unsound recognizer has a false-positive rate of ρ—*i.e.*, on average, each unsound recognizer outputs about ρM_n false-positives (in addition to the M_n true-positives instances). In this case, there are $\frac{((1+\rho)M_n)!}{(\rho M_n)!M_n!}$ ways to choose M_n instances from the $(1+\rho)M_n$ that were recognized by *each* of the U unsound recognizers. Therefore, across all recognizers, the PTPSets function call at line 6.1(b) produces a set containing about $\left(\frac{((1+\rho)M_n)!}{(\rho M_n)!M_n!}\right)^U$ members. Multiplying these two bounds together, we have that:

$$J_n \approx K! \left(\frac{((1+\rho)M_n)!}{(\rho M_n)!M_n!}\right)^U . \tag{6.8}$$

In summary, J_n (and hence $\prod_n J_n$) is potentially enormous. For example, if each recognizer has a false positive rate of 10% (*i.e.*, $\rho = 0.1$), there are 10 tuples per page ($M = 10$) and 4 unsoundly-recognized attributes ($U = 4$), then $\prod_n J_n = 14641$ possible labels must be considered.[3]

We conclude that a straightforward implementation of Corrob is impractical because its output contains so many labels. Therefore, we implemented a modified version of Corrob, which for clarity we will refer to as Corrob*. The two algorithms differ in three major ways, which can be characterized as follows:

1. The first modification is to provide *additional input* to the algorithm, to make the problem easier.
2. The second modification involves using *greedy heuristics*, yielding outputs that are (strictly speaking) incorrect, though (as our experiments demonstrate) useful anyway.
3. The third modification involves using *domain-specific heuristics*.

6.6.1 Additional input: Attribute ordering

Recall that Corrob determines which attribute orderings "$\prec$" are consistent with the recognized instances, and returns labels corresponding to each such consistent ordering.

Our first simplification is to provide Corrob* with the correct attribute ordering "$\prec$" as an additional input. As the complexity analysis above indicates, enumerating all $K!$ possible orderings is expensive; see Corrob's line 6.1(c). Therefore, we decided to eliminate this (admittedly interesting) aspect of Corrob's functionality. In terms

[3] Note that the parameter ρ is introduced only to simplify this complexity analysis; ρ plays no role in the Corrob algorithm.

of the analysis above, this simplification removes the $K!$ term from the estimate of each J_n value.

6.6.2 Greedy heuristic: Strongly-ambiguous instances

Corrob's enumeration of all attribute orderings is one reason it performs poorly. The second source of difficulty is that the algorithm's PTPSets subroutine enumerates all possible subsets of the instances recognized by unsound recognizers; see Corrob's line 6.1(b).

However, we can improve this aspect of Corrob by categorizing the unsoundly-recognized instances into three groups: those that are *unambiguous*, those that are *weakly ambiguous*, and those that are *strongly ambiguous.* For example, consider a library of three recognizers which respond as follows:

$\mathcal{R}_{\text{CTRY}}$ *perfect*
$\langle 15, 20\rangle$
$\langle 35, 40\rangle$
$\langle 55, 60\rangle$

$\mathcal{R}_{\text{CODE}}$ *unsound*	
$\langle 22, 27\rangle$	U
$\langle 42, 47\rangle$	W
$\langle 41, 46\rangle$	W
$\langle 62, 67\rangle$	S
$\langle 65, 70\rangle$	S

$\mathcal{R}_{\text{CAP}}$ *unsound*	
$\langle 29, 34\rangle$	U
$\langle 49, 54\rangle$	U
$\langle 69, 74\rangle$	S
$\langle 72, 78\rangle$	S

We have annotated the unsoundly-recognized instances (*i.e.*, the CODE and CAP instances) as follows: "U" indicates that the instance is unambiguous; "W", weakly ambiguous; and "S", strongly ambiguous.

To understand these annotations, consider the output of Corrob:

$$\left\{\begin{array}{cc}
\begin{array}{|c|c|c|}\hline \text{CTRY} & \text{CODE} & \text{CAP} \\ \hline \langle 15,20\rangle & \langle 22,27\rangle & \langle 29,34\rangle \\ \langle 35,40\rangle & \langle 42,47\rangle & \langle 49,54\rangle \\ \langle 55,60\rangle & \langle 62,67\rangle & \langle 69,74\rangle \\ \hline \end{array}, &
\begin{array}{|c|c|c|}\hline \text{CTRY} & \text{CODE} & \text{CAP} \\ \hline \langle 15,20\rangle & \langle 22,27\rangle & \langle 29,34\rangle \\ \langle 35,40\rangle & \langle 41,46\rangle & \langle 49,54\rangle \\ \langle 55,60\rangle & \langle 62,67\rangle & \langle 69,74\rangle \\ \hline \end{array}, \\[2em]
\begin{array}{|c|c|c|}\hline \text{CTRY} & \text{CODE} & \text{CAP} \\ \hline \langle 15,20\rangle & \langle 22,27\rangle & \langle 29,34\rangle \\ \langle 35,40\rangle & \langle 42,47\rangle & \langle 49,54\rangle \\ \langle 55,60\rangle & \langle 62,67\rangle & \langle 72,78\rangle \\ \hline \end{array}, &
\begin{array}{|c|c|c|}\hline \text{CTRY} & \text{CODE} & \text{CAP} \\ \hline \langle 15,20\rangle & \langle 22,27\rangle & \langle 29,34\rangle \\ \langle 35,40\rangle & \langle 41,46\rangle & \langle 49,54\rangle \\ \langle 55,60\rangle & \langle 62,67\rangle & \langle 72,78\rangle \\ \hline \end{array}, \\[2em]
\begin{array}{|c|c|c|}\hline \text{CTRY} & \text{CODE} & \text{CAP} \\ \hline \langle 15,20\rangle & \langle 22,27\rangle & \langle 29,34\rangle \\ \langle 35,40\rangle & \langle 42,47\rangle & \langle 49,54\rangle \\ \langle 55,60\rangle & \langle 65,70\rangle & \langle 72,78\rangle \\ \hline \end{array}, &
\begin{array}{|c|c|c|}\hline \text{CTRY} & \text{CODE} & \text{CAP} \\ \hline \langle 15,20\rangle & \langle 22,27\rangle & \langle 29,34\rangle \\ \langle 35,40\rangle & \langle 41,46\rangle & \langle 49,54\rangle \\ \langle 55,60\rangle & \langle 65,70\rangle & \langle 72,78\rangle \\ \hline \end{array}
\end{array}\right\}. \tag{6.9}$$

We begin by observing that these six labels are in fact the output of Corrob, because they are the only labels that satisfy Consistent? among the

$$\underbrace{\left(\frac{5!}{3!2!}\right)}_{|\mathcal{R}_{\text{CODE}}(P)|=5} \times \underbrace{\left(\frac{4!}{3!1!}\right)}_{|\mathcal{R}_{\text{CAP}}(P)|=4} = 40$$

possibilities considered (*i.e.*, among the ways to choose $M = 3$ instances each from the sets $\mathcal{R}_{\text{CODE}}(P)$ and $\mathcal{R}_{\text{CAP}}(P)$). (We omit the details.)

These annotations can be understood as follows:

- Note that the CODE instance $\langle 22, 27\rangle$ occurs in *every* label in Equation 6.9. Instance $\langle 22, 27\rangle$ is *unambiguous* because, though it was recognized by an unsound recognizer, $\langle 22, 27\rangle$ is necessarily a true-positive, by virtue of the other instances. Similarly, the CAP instances $\langle 29, 34\rangle$ and $\langle 49, 54\rangle$ are also unambiguous.

  In summary, an unsoundly-recognized instance is unambiguous if and only if it occurs in every label output by Corrob.

- Now consider the CODE instance $\langle 65, 70\rangle$. It occurs in some but not all of the labels in Equation 6.9, so $\langle 65, 70\rangle$ is not unambiguous. But if we look more carefully, we discover a relationship between the CODE attribute $\langle 65, 70\rangle$ and instances of the CAP attribute: $\langle 65, 70\rangle$ occurs in a given label only when the CAP instance $\langle 72, 78\rangle$ occurs in the same label. We say that $\langle 65, 70\rangle$ is *strongly ambiguous*, because whether it is included in a label depends on the instances are selected for *other* attributes. The word "strongly" is meant to stress that resolving the ambiguity requires examining a (potentially large) portion of the label.

  Of course, since $\langle 65, 70, \textsc{code}\rangle$[4] depends on $\langle 72, 78, \textsc{cap}\rangle$, then $\langle 72, 78, \textsc{cap}\rangle$ depends on $\langle 65, 70, \textsc{code}\rangle$, and so CAP's instance $\langle 72, 78\rangle$ is strongly ambiguous as well.

  Moreover, note that $\langle 65, 70, \textsc{code}\rangle$ "competes with" instance $\langle 62, 67, \textsc{code}\rangle$, in that both "belong to" the same cell of the label. In the true label there is exactly one instance per cell, and thus these two instances are mutually exclusive. And since $\langle 65, 70, \textsc{code}\rangle$ is strongly ambiguous, we say that $\langle 62, 67, \textsc{code}\rangle$ is strongly ambiguous too. By similar reasoning we have that $\langle 69, 74, \textsc{cap}\rangle$ is strongly ambiguous, because it is mutually exclusive with the strongly ambiguous instance $\langle 72, 78, \textsc{cap}\rangle$.

  More precisely, an instance is strongly ambiguous if it interacts with another unsoundly-recognized attribute in the sense just discussed, or if it mutually excludes a strongly ambiguous instance.

  Finally, note that the fact that strong ambiguity propagates from one instance to another means that strong ambiguity induces a partition on a label's strongly ambiguous instances: two strongly ambiguous instances belong to the same

[4] Recall that to emphasize the attribute to which an instance belongs, we use the notation $\langle b, e, F_k\rangle$ to indicate that $\langle b, e\rangle$ is an instance of attribute F_k.

partition element iff they are connected by a chain of strongly ambiguous relationships. We call each such partition element a *cluster* of strongly ambiguous instances. In the example, the clustering is trivial, with all of the strongly ambiguous instances belonging to the single cluster

$$\{\langle 65, 70, \textsc{code}\rangle, \langle 62, 67, \textsc{code}\rangle, \langle 69, 74, \textsc{cap}\rangle, \langle 72, 78, \textsc{cap}\rangle\}\,.$$

In general, each cluster contains one or more instances from each of several attributes.

- Finally, note that $\langle 42, 47, \textsc{code}\rangle$ and $\langle 41, 46, \textsc{code}\rangle$ are neither unambiguous nor strongly ambiguous. These two instances are mutually exclusive, but the choice of including one or the other in a particular label can be made independently from choices for the other attributes. Therefore, we say that $\langle 42, 47, \textsc{code}\rangle$ and $\langle 41, 46, \textsc{code}\rangle$ are *weakly ambiguous*.

  To summarize, an instance is weakly ambiguous if it is neither unambiguous nor strongly ambiguous.

We are now finally in a position to describe Corrob*'s greedy heuristic. Whereas Corrob includes *all* unsoundly-recognized instances, Corrob* outputs all unambiguous and weakly ambiguous instances, but only some of the strongly ambiguous instances.

Specifically, for each cluster of strongly ambiguous instances, Corrob* arbitrarily selects one attribute from the cluster's instances, and then discards all instances in the cluster except those of the selected attribute. Note that by discarding all but one attribute's instances, that attribute's instances are rendered weakly (rather than strongly) ambiguous.

In the example, Corrob* would create a set of labels based on the following instances:

necessarily true-positives:	$\langle 15, 20, \textsc{ctry}\rangle$, $\langle 35, 40, \textsc{ctry}\rangle$, $\langle 55, 60, \textsc{ctry}\rangle$
unambiguous instances:	$\langle 22, 27, \textsc{code}\rangle$, $\langle 29, 34, \textsc{cap}\rangle$, $\langle 49, 54, \textsc{cap}\rangle$
weakly ambiguous instances:	$\langle 42, 47, \textsc{code}\rangle$, $\langle 41, 46, \textsc{code}\rangle$
strongly ambiguous instances:	*either* $\langle 62, 67, \textsc{code}\rangle$, $\langle 65, 70, \textsc{code}\rangle$ †
	or $\langle 69, 74, \textsc{cap}\rangle$, $\langle 72, 78, \textsc{cap}\rangle$

The first three items listed have already been discussed; the fourth item illustrates Corrob^*'s greediness. In the first case (marked "†"), the instances $\langle 69, 74, \textsc{cap}\rangle$ and $\langle 72, 78, \textsc{cap}\rangle$ are simply discarded from further consideration. In the second case $\langle 62, 67, \textsc{code}\rangle$, $\langle 65, 70, \textsc{code}\rangle$ are discarded. Let us emphasize that Corrob^*'s choice of which attribute's instances to select is made arbitrarily (on no principled basis), and greedily (once the choice is made it is never reconsidered).

Suppose Corrob^* chooses the first set of strongly ambiguous instances, marked "†" above. In this case, Corrob^* produces the following four labels as output:

$$\left\{ \begin{array}{c} \begin{array}{|c|c|c|} \hline \textsc{ctry} & \textsc{code} & \textsc{cap} \\ \hline \langle 15,20\rangle & \langle 22,27\rangle & \langle 29,34\rangle \\ \langle 35,40\rangle & \langle 42,47\rangle & \langle 49,54\rangle \\ \langle 55,60\rangle & \langle 62,67\rangle & ? \\ \hline \end{array} , \begin{array}{|c|c|c|} \hline \textsc{ctry} & \textsc{code} & \textsc{cap} \\ \hline \langle 15,20\rangle & \langle 22,27\rangle & \langle 29,34\rangle \\ \langle 35,40\rangle & \langle 41,46\rangle & \langle 49,54\rangle \\ \langle 55,60\rangle & \langle 62,67\rangle & ? \\ \hline \end{array} , \\ \\ \begin{array}{|c|c|c|} \hline \textsc{ctry} & \textsc{code} & \textsc{cap} \\ \hline \langle 15,20\rangle & \langle 22,27\rangle & \langle 29,34\rangle \\ \langle 35,40\rangle & \langle 42,47\rangle & \langle 49,54\rangle \\ \langle 55,60\rangle & \langle 65,70\rangle & ? \\ \hline \end{array} , \begin{array}{|c|c|c|} \hline \textsc{ctry} & \textsc{code} & \textsc{cap} \\ \hline \langle 15,20\rangle & \langle 22,27\rangle & \langle 29,34\rangle \\ \langle 35,40\rangle & \langle 41,46\rangle & \langle 49,54\rangle \\ \langle 55,60\rangle & \langle 65,70\rangle & ? \\ \hline \end{array} \end{array} \right\} . \tag{6.10}$$

Note that since the instances that were to have occupied the bottom-left cell have been discarded, Corrob^* must fill this cell with "?" in each label. This "?" symbol demonstrates that Corrob^*'s greedy discarding of strongly ambiguous instances means that it loses information about a page's true label.

However, this loss is repaid with a substantial computational gain. Specifically, note that we can compactly represent the entire set in Equation 6.10 by the following structure:

$$\begin{array}{|c|c|c|} \hline \textsc{ctry} & \textsc{code} & \textsc{cap} \\ \hline \langle 15,20\rangle & \langle 22,27\rangle & \langle 29,34\rangle \\ \langle 35,40\rangle & \langle 41,46\rangle \oplus \langle 42,47\rangle & \langle 49,54\rangle \\ \langle 55,60\rangle & \langle 62,67\rangle \oplus \langle 65,70\rangle & ? \\ \hline \end{array} \tag{6.11}$$

The exclusive-or "$\oplus$" symbols indicate that *exactly one* instance in these cells can occur in any given label. This structure is a compact representation of Equation 6.10 because there are four ways to choose one of the two elements in each of the two "$\oplus$" cells; each combination corresponds to a member of the set in Equation 6.10.

Note that Corrob^* can always represent its set of output labels using such a compact representation. This property holds because, with no strongly ambiguous instances, decisions about how to resolve any remaining ambiguity (*i.e.*, how to choose one instance from each weakly ambiguous set) can always be made regardless of how decisions are made in unrelated parts of the label.

We can summarize this second simplification in the faster implementation Corrob^* of the slow Corrob algorithm as follows. First, we showed how to categorize unsoundly-recognized instances as either unambiguous, weakly ambiguous, or strongly ambiguous. We then suggested a simple technique—discarding all but one attribute's instances from each cluster of strongly ambiguous instances—which always permits a compact representation of the entire set of Corrob^*'s output labels. In essence, Corrob^* treats unsound recognizers as if they were incomplete whenever the instances they output are strongly ambiguous, by "pretending" it never observed the instances in the first place.

Of course this modification to Corrob means that Corrob^* is not formally correct (according to the initial definition of the labeling problem). Specifically, Corrob^*'s output might contain "?" symbols in cells that ought to be occupied. However, just as $\mathsf{Generalize}^{\mathrm{noisy}}_{\mathrm{HLRT}}$ can learn in the face of incomplete recognizers, so too can it learn when some of the strongly ambiguous instances have been discarded. While not formally correct, Corrob^*'s output is "good enough" that it can be used by $\mathsf{Generalize}^{\mathrm{noisy}}_{\mathrm{HLRT}}$ anyway.

To review, the first simplification to Corrob is to provide the ordering "$\prec$" in addition to the page P and the recognizer library Δ. This second modification means that the output of the function call $\mathsf{Corrob}^*(P, \Delta, \text{"}\prec\text{"})$ is a compact label structure, such as in Equation 6.11. As before, $\mathsf{Generalize}^{\mathrm{noisy}}_{\mathrm{HLRT}}$ now must iterate over all the ways to resolve the remaining ambiguities—*i.e.*, all ways to choose one instance from each "$\oplus$" group. The advantage of the compact representation now becomes clear: we can save time by enumerating labels only as they are needed by $\mathsf{Generalize}^{\mathrm{noisy}}_{\mathrm{HLRT}}$. However, in which order should $\mathsf{Generalize}^{\mathrm{noisy}}_{\mathrm{HLRT}}$ examine these labels (see line 6.2(a) of the algorithm)? The third modification to Corrob addresses this question.

6.6.3 Domain-specific heuristic: Proximity ordering

In the example developed in this section, $\mathsf{Generalize}^{\mathrm{noisy}}_{\mathrm{HLRT}}$'s line 6.2(a) eventually might examine all four labels to which the compact representation in Equation 6.11 is equivalent. Ideally, we want the true label to be examined first, so that $\mathsf{Generalize}^{\mathrm{noisy}}_{\mathrm{HLRT}}$ can ignore the rest. How should $\mathsf{Generalize}^{\mathrm{noisy}}_{\mathrm{HLRT}}$ select among the 4! possible orders in which to examine these four labels?

To answer this question, Corrob* employs a simple heuristic, which we call *proximity ordering*. Consider the second row of the compact representation listed in Equation 6.11. We need to choose one instance from $\langle 41, 46\rangle \oplus \langle 42, 47\rangle$. Note that instance $\langle 41, 46\rangle$ is closer to the interval $\langle 35, 40\rangle$ than $\langle 42, 47\rangle$—*i.e.*, $40 < 41 < 42$. Proximity ordering simply involves trying $\langle 41, 46\rangle$ before $\langle 42, 47\rangle$ on the basis of this distance relationship.[5]

Similarly, in the choice of one instance from $\langle 62, 67\rangle \oplus \langle 65, 70\rangle$, proximity ordering tries $\langle 62, 67\rangle$ before $\langle 65, 70\rangle$, because $60 < 62 < 65$.

Of course, when $\mathsf{Generalize}^{\mathrm{noisy}}_{\mathrm{HLRT}}$ queries this compact representation for a label, the result must involve a selection of one instance from *each* of the "$\oplus$" groups. Proximity ordering in this more general sense states that the labels are to be visited in order of decreasing total proximity.

For example, when applied to the representation in Equation 6.11, the proximity

[5] To be more accurate, we should refer to this heuristic as *left-proximity ordering*, because it ignores proximity to instances on the right.

ordering heuristic suggests visiting in the following order the four labels listed in Equation 6.10:

1.

CTRY	CODE	CAP
$\langle 15, 20\rangle$	$\langle 22, 27\rangle$	$\langle 29, 34\rangle$
$\langle 35, 40\rangle$	$\langle 41, 46\rangle$	$\langle 49, 54\rangle$
$\langle 55, 60\rangle$	$\langle 62, 67\rangle$	?

2.

CTRY	CODE	CAP
$\langle 15, 20\rangle$	$\langle 22, 27\rangle$	$\langle 29, 34\rangle$
$\langle 35, 40\rangle$	$\langle 41, 46\rangle$	$\langle 49, 54\rangle$
$\langle 55, 60\rangle$	$\langle 65, 70\rangle$	?

3.

CTRY	CODE	CAP
$\langle 15, 20\rangle$	$\langle 22, 27\rangle$	$\langle 29, 34\rangle$
$\langle 35, 40\rangle$	$\langle 42, 47\rangle$	$\langle 49, 54\rangle$
$\langle 55, 60\rangle$	$\langle 62, 67\rangle$	?

4.

CTRY	CODE	CAP
$\langle 15, 20\rangle$	$\langle 22, 27\rangle$	$\langle 29, 34\rangle$
$\langle 35, 40\rangle$	$\langle 42, 47\rangle$	$\langle 49, 54\rangle$
$\langle 55, 60\rangle$	$\langle 65, 70\rangle$	?

(Note that proximity is a relative rather than metric: the proximity ordering heuristic is indifferent to the order between the second and third labels.)

Proximity ordering is clearly a domain-specific heuristic. As such, it might behave well (*i.e.*, force early consideration of a label matching the true label) for some labeling problems, and poorly for others. In Chapter 7, we demonstrate that this heuristic works surprisingly well for several actual Internet information resources.

6.6.4 Performance of Corrob*

At this point, the natural questions to ask are: what is the performance of the Corrob* algorithm, and how large is the improvement over Corrob?

Analytical answers (*e.g.*, asymptotic complexity bound) to these questions would be very interesting. Unfortunately, such bounds are difficult to derive. The problem is that the number of labels consistent with any given example (the main parameter governing the algorithm's running time) depends on the specific mistakes made by the recognizer library.

Therefore, we have taken an empirical approach to verifying whether Corrob* runs quickly. Section 7.3 reports that, for several actual Internet resources, Corrob* represents a small fraction of the overall running time of our wrapper construction system.

6.7 Recognizers

This chapter has made extensive use of the notion of recognizers. We have assumed that a library of possibly-noisy recognizers is provided as input, discussed techniques for corrobating the evidence provided by such recognizers, and shown how to extend the algorithms and analysis from Chapter 4 to robustly learn HLRT wrappers in the face of the incorrectly labeled pages.

However, we have not actually discussed recognizers *per se*. The reason is simply that this thesis is concerned with wrapper induction, rather than with the details of any particular class of recognizer. Nevertheless, in this section we address several important issues related to recognizers.

Multiple recognizers. First, recall that the recognizer library is assumed to contain exactly one recognizer per attribute. In fact this is an unimportant restriction. The algorithms, analysis, and technqiues discussed in this chapter can all be extended to handle multiple recognizers per attribute. Importantly, this holds even when the recognizers make different kinds of mistakes. For example, the Corrob algorithm can easily be extended so that two country recognizers can be provided, one unsound and the other incomplete.

Perfect recognizers? Assumption 6.1's requirement that the recognizer library contains at least one perfect recognizer might seem too strong. However, we believe there are two reasons to be optimistic.

First, many of the attributes which we want to be able to extract *can* be recognized perfectly. For example, regular expressions can be used to perfectly match attributes such as electronic mail addresses, URLs, ISBN numbers, Internet IP addresses, credit card numbers, and US telephone numbers, ZIP codes and states. It is certainly true that an adversary can usually defeat such a simple mechanism. However, note that a recognizer claiming to be "perfect" need only behave perfectly with respect to the information resources to which it is applied, and such resources are unlikely to engage in a conspiracy with the recognizer's adversary.

Second, though this thesis has largely concentrated on *automatic* wrapper construction, the techniques developed are compatible with a related (though more modest) goal: *semi-automatic* wrapper construction. Therefore, if a perfect recognizer is needed for a particular attribute, then there is always the possibility of asking a person, who (presumably) always give the correct answer.

Unreliable recognizers? Assumption 6.1 also prohibits unreliable recognizers. Corrob requires this stipulation for two reasons. First, specifying the operation of Corrob for unrealiable recognizers is possible but somewhat more complicated. Therefore, in the interest of simplicity, we ignored unrealiable recognizers.

However, unreliable recognizers also introduce a tremendous complexity blowup into the corroboration process. The basic problem is that *every* instance returned by an unsound recognizer is suspect. (In contrast, *none* of an incomplete recognizer's instances, and *all but* M_n of an unsound recognizer's instances, are suspect.) In concrete terms, this means that with unsound recognizers the PTPSets subroutine in Figure 6.1 returns vastly more sets of possibly true-positive instances, compared to libraries without unsound recognizers. To use the notation introduced earlier, the problem with unreliable recognizers is that they greatly increase J_n (the number of labels Corrob* outputs for page P_n; see Equation 6.8).

Of course, we have seen that even without unreliable recognizers, the Corrob algorithm has a high computational complexity. However, the techniques mentioned in Section 6.6 to alleviate these problems rely on the assumption of no unreliable recognizers. We leave as open the issues related to efficient corroboration algorithms that do not require Assumption 6.1.

One-sided error? Since we essentially disallow unreliable recognizers, our recognizer noise model requires that recognizers make one-sided error—*i.e.*, either false positives, or false negatives, but not both. How realistic is this restriction? Let us mention several reasons for being optimistic.

First of all, many techniques for implementing recognizers (see below for more details) naturally give rise to one-sided errors. For example, if one needs a recognizer for company names, then an index of companies (from the Fortune 500 list, for example) can be used to implement an incomplete company-name recognizer. On the other hand, consider a regular-expression–based US ZIP+4 recognizer that searches for the pattern `[0-9][0-9][0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]`. This recognizer is most likely unsound, since it might recognize non-ZIP+4 instances (*e.g.*, fragments of credit-card numbers). On the other hand, this recognizer will never incorrectly miss a ZIP+4 code.

Second, we note that in some cases, it might be feasible to decompose a recognizer that gives two-sided errors—*i.e.*, an unreliable recognizer—into two recognizers, each having one-sided error: one that is incomplete, and another that is unsound. If this decomposition is possible, then we will have effectively included an unreliable recognizer into the library without suffering the expected computational cost.

Off-the-shelf recognition technology. Finally, let us briefly describe several existing technologies that might be useful for implementing recognizers.

First, simple regular-expression mechanisms can be surpisingly powerful for implementing recognizers, even for relatively structured and complicated attributes. For example, using lists of abbreviations obtained from the US Postal Service and related knowledge, we have designed a regular expression that does a very good job of recognizing US street addresses; the patten is about 15,000 characters long. While regular expressions are certainly simple, they should not be understated.

Second, there is a large body of research in the natural-language processing community concerned with identifying classes of entities in free text. For example, the "Named Entity" task at the MUC-6 conference [ARPA 95] involves locating people's names, company names, dates, times, locations, and so forth. This research has matured to the point that high-quality commercial people-name recognizers are now available—examples include the Carnegie Group's "NameFinder" system [www.cgi.com], and "SSA-NAME" [www.searchsoftware.com].

Third, many different entities—government agencies, companies, *etc.*—are involved with compiling specialized databases, indices, encylopedias, dictionaries, catalogs, surveys and so forth; examples include "Books in Print" and "www.imdb.com". Consider the following simple example: a telephone directory is essentially a list of people and businesses, and their addresses and telephone numbers; an incomplete address recognizer or name recognizer can be readily built by consulting such a list.

Of course, extracting the content from such resources for the purpose of building recognizers leads to a "meta wrapper induction" problem. This idea leads to a fourth strategy for building recognizers. Once a single information resource has been successfully "wrapped", the resource can be "mined" [Etzioni 96b] for additional information that can subsequently be used to build recognizers for other resources. For example, once one telephone directory has been wrapped (with assistance from a person, perhaps) then the resource can be queried when learning a wrapper for a second telephone directory. This "bootstrapping" approach might prove useful as more and more sites providing information related to any particular topic come online.

6.8 Summary

In this chapter we introduced *corroboration*, a technique for labeling the example pages need by the Induce generic inductive learning algorithm. The main results of this chapter are a precise statement of the labeling problem, an algorithm Corrob for solving an interesting special case of this problem.

Chapter 7

EMPIRICAL EVALUATION

7.1 Introduction

In this thesis we have proposed a collection of related techniques—wrapper induction for several wrapper classes, PAC analysis, and corroboration—designed to address the problem of automatically constructing wrappers for semi-structured information resources. How well do our techniques work? Our approach to evaluation is thoroughly empirical. In this chapter we describe several experiments that measure our system's performance on actual Internet resources

7.2 Are the six wrapper classes useful?

In Chapters 4 and 5, we identified six wrapper classes: LR, HLRT, OCLR, HOCLRT, N-LR and N-HLRT. Learnability aside, an important issue is whether these classes are useful for handling the actual information resources we would like our software agents to use. That is, before addressing *how* to learn wrappers, we must ask whether we should *bother* to do so. Our approach to answering this question was to conduct a survey. In a nutshell, we examined a large collection of resources, and found that the majority (70% in total) of the resources can be covered by our six wrapper classes.

To maintain objectivity, we selected the resources from an independently collected index. The Internet site "www.search.com" maintains an index of 448 Internet resources. A wide variety of topics are included: from the Abele Owners' Network [www.owner.com] ("over 30,000 properties nationwide; the national resource for homes sold by owner") to Zipper [www.voxpop.org/zipper] ("find the name of your representative or senator, along with the address, phone number, email, and Web page").[1] While the Internet obviously contains more than 448 sites, we expect that this index is representative of sites that software agents might use.

To perform the survey, we first randomly selected 30 resources (*i.e.*, $\frac{30}{448} = 6.7\%$) from www.search.com's index; Figure 7.1 lists the surveyed sites. The figure also shows the number of attributes (K) extracted from each; K ranges from two to eighteen.

Next, for each of the thirty sites, we gathered the responses to ten sample queries. The queries were chosen by hand. We choose queries that were appropriate to the resource in question. For example, for the first site in Figure 7.1 (a computer hardware vendor), the sample queries were "pentium pro", "newton", "hard disk", "cache memory", "macintosh", "server", "mainframe", "zip", "backup" and "monitor". Our intent was to solicit normal responses, rather than unusual responses (*e.g.*, error responses, pages containing no data, *etc.*).[2]

To complete the survey, we determined how to fill in the thirty-by-six matrix, indicating for each resource whether it can be handled by each of the wrapper classes. To fill in this matrix, we labeled the examples by hand, and then used the $\mathsf{Generalize}_{\mathcal{W}}$ algorithm to try to learn a wrapper in class $\mathcal{W}$ that is consistent with the resource's ten examples. Note that (as discussed in Section 6.5.1 for the HLRT class) it is trivial to modify $\mathsf{Generalize}_{\mathcal{W}}$ so that it detects if no consistent wrapper exists in class $\mathcal{W}$.

Our results are listed in Figure 7.2; "$\surd$" indicates that the given resource can be wrapped by the given wrapper class, while "×" indicates that there does not exist a wrapper in the class consistent with the collected examples.

Figure 7.3 shows a table summarizing Figure 7.2. Each line in the table indicates

[1] The site www.search.com is constantly updating its index. Our survey was conducted in July 1997; naturally, some of the sites might have disappeared or changed significantly since then.

[2] While learning to handle such exceptional situations is important, our work does not address this problem; see [Doorenbos et al. 97] for some interesting progress in this area.

	resource	URL	K
1	Computer ESP	`http://www.computeresp.com`	4
2	CNN/Time AllPolitics Search	`http://allpolitics.com/`	4
3	Film.com Search	`http://www.film.com/admin/search.htm`	6
4	Yahoo People Search: Telephone/Address	`http://www.yahoo.com/search/people/`	4
5	Cinemachine: The Movie Review Search Engine	`http://www.cinemachine.com/`	2
6	PharmWeb's World Wide List of Pharmacy Schools	`http://www.pharmweb.net/`	13
7	TravelData's Bed and Breakfast Search	`http://www.ultranet.com/biz/inns/search-form.html`	4
8	NEWS.COM	`http://www.news.com/`	3
9	Internet Travel Network	`http://www.itn.net/`	13
10	Time World Wide	`http://pathfinder.com/time/`	4
11	Internet Address Finder	`http://www.iaf.net/`	6
12	Expedia World Guide	`http://www.expedia.com/pub/genfts.dll`	2
13	thrive@pathfinder	`http://pathfinder.com/thrive/index.html`	4
14	Monster Job Newsgroups	`http://www.monster.com/`	3
15	NewJour: Electronic Journals & Newsletters	`http://gort.ucsd.edu/newjour/`	2
16	Zipper	`http://www.voxpop.org/zipper/`	11
17	Coolware Classifieds Electronic Job Guide	`http://www.jobsjobsjobs.com`	2
18	Ultimate Band List	`http://ubl.com`	2
19	Shops.Net	`http://shops.net/`	5
20	Democratic Party Online	`http://www.democrats.org/`	6
21	Complete Works of William Shakespeare	`http://the-tech.mit.edu/Shakespeare/works.html`	5
22	Bible (Revised Standard Version)	`http://etext.virginia.edu/rsv.browse.html`	3
23	Virtual Garden	`http://pathfinder.com/vg/`	3
24	Foreign Languages for Travelers Site Search	`http://www.travlang.com/`	4
25	U.S. Tax Code On-Line	`http://www.fourmilab.ch/ustax/ustax.html`	2
26	CD Club Web Server	`http://www.cd-clubs.com/`	5
27	Expedia Currency Converter	`http://www.expedia.com/pub/curcnvt.ddll`	6
28	Cyberider Cycling WWW Site	`http://blueridge.infomkt.ibm.com/bikes/`	3
29	Security APL Quote Server	`http://qs.secapl.com/`	18
30	Congressional Quarterly's On The Job	`http://voter96.cqalert.com/cq_job.htm`	8

Figure 7.1: *The information resources that we surveyed to measure wrapper class coverage.*

							region in	region in
resource	LR	HLRT	OCLR	HOCLRT	N-LR	N-HLRT	Fig. 5.1	Fig. 5.3
1	√	√	√	√	×	×	*(E)*	*(S)*
2	×	×	×	×	×	×	*(A)*	*(J)*
3	√	√	√	√	×	×	*(E)*	*(S)*
4	√	√	√	√	√	√	*(E)*	*(T)*
5	√	√	√	√	×	√	*(E)*	*(N)*
6	×	×	×	×	×	×	*(A)*	*(J)*
7	×	×	×	×	×	√	*(A)*	*(P)*
8	√	√	√	√	×	√	*(E)*	*(N)*
9	×	×	×	×	×	×	*(A)*	*(J)*
10	√	√	√	√	×	×	*(E)*	*(M)*
11	×	×	×	×	×	×	*(A)*	*(J)*
12	×	√	×	√	×	√	*(C)*	*(O)*
13	√	√	√	√	×	×	*(E)*	*(M)*
14	×	√	×	√	×	√	*(C)*	*(O)*
15	√	√	√	√	×	√	*(E)*	*(N)*
16	×	×	×	×	×	×	*(A)*	*(J)*
17	×	×	×	×	×	√	*(A)*	*(P)*
18	×	×	×	×	×	√	*(A)*	*(P)*
19	√	√	√	√	√	√	*(E)*	*(T)*
20	√	√	√	√	√	√	*(E)*	*(T)*
21	×	×	×	×	×	×	*(A)*	*(J)*
22	√	√	√	√	×	√	*(E)*	*(N)*
23	√	√	√	√	×	√	*(E)*	*(N)*
24	×	×	×	×	×	×	*(A)*	*(J)*
25	√	√	√	√	×	√	*(E)*	*(N)*
26	×	×	×	×	×	×	*(A)*	*(J)*
27	√	√	√	√	×	√	*(E)*	*(N)*
28	√	×	√	×	√	×	*(G)*	*(R)*
29	×	×	×	×	×	×	*(A)*	*(J)*
30	√	√	√	√	×	×	*(E)*	*(M)*
total	16	17	16	17	4	15		

Figure 7.2: *The results of our coverage survey.*

wrapper class(es)	coverage (%)
LR ∪ HLRT ∪ OCLR ∪ HOCLRT ∪ N-LR ∪ N-HLRT	70
LR ∪ HLRT ∪ OCLR ∪ HOCLRT	60
LR ∪ OCLR	53
LR	53
OCLR	53
HLRT ∪ HOCLRT	57
HLRT	57
HOCLRT	57
N-LR ∪ N-HLRT	53
N-LR	13
N-HLRT	50
N-LR ∪ N-HLRT when not LR ∪ HLRT ∪ OCLR ∪ HOCLRT	25

Figure 7.3: *A summary of Figure 7.2.*

the coverage of one or more wrapper classes. For example, the first line indicates that $\frac{21}{30} = 70\%$ of the surveyed sites can be handled by one or more of the six wrapper classes.

Figure 7.3 reports the coverage for several groups of wrapper classes. The groups are organized hierarchically: first we distinguish between wrappers for tabular and nested resources; we then partition the tabular wrapper classes according to the relative expressiveness results described by Theorem 5.1 (Section 5.2.3). We conclude that, with the exception of N-LR, the wrapper classes we have identified are indeed useful for handling actual Internet resources.

Notice that the worst performing wrapper classes are N-LR and N-HLRT. Recall that we introduced the N-LR and N-HLRT wrapper classes in order to handle the resources whose content exhibited a nested rather than tabular structure. The last line of the table above measures how successful we were: we find that N-LR and N-HLRT cover 25% of the resources that the other four classes can not handle. We conclude that, despite their relatively poor showing overall, N-LR and N-HLRT do indeed provide expressiveness not available with the other four classes.

Finally, recall from Chapter 5 the discussion of relative expressiveness, the extent to which wrappers in one class can mimic those in another. We described the ways that the six classes are related: Figure 5.1 and Theorem 5.1 for LR, HLRT, OCLR and HOCLRT; and Figure 5.3 and Theorem 5.10 for LR, HLRT, N-LR and N-HLRT.

Figure 7.2 indicates the regions in Figures 5.1 and 5.3 to which each of the surveyed resources belongs. For the tabular classes (for LR, HLRT, OCLR and HOCLRT), the surveyed sites are located in several of the regions in Figure 5.1, though the surveyed sites do not show the expressiveness differences between LR and OCLR, or between HLRT and HOCLRT. For the nested classes (N-LR and N-HLRT), the surveyed sites are distributed in most of the regions in Figure 5.1. We conclude that our theoretical results concerning relative expressiveness reflect reasonably well the differences observed among actual Internet resources.

7.3 Can HLRT *be learned quickly?*

In the previous section, we saw that it would be useful to learn wrappers in the six classes described in this thesis, because these six classes can in fact wrap numerous actual Internet resources. We now go on to ask: how well does our automatic learning system work? To answer this question, we measured the performance of our system when learning the HLRT wrapper class.

We tested our HLRT learning algorithm on 21 actual Internet resources. Seventeen of the surveyed resources listed in Figure 7.1 were chosen; namely, those resources that can be handled by HLRT (*i.e.*, whose second column is marked '$\surd$' in Figure

7.2): sites 1, 3, 4, 5, 8, 10, 12, 13, 14, 15, 19, 20, 22, 23, 25, 27 and 30.

Four additional resources were examined:

OKRA [okra.ucr.edu], an email address locator service.[3] The query is a person's name; $K = 4$ attributes are extracted: name, email address, database entry date, and confidence score. Responses to 252 queries were collected; the queries were randomly selected from a collection of people's last names gathered from newsgroup posts (*e.g.*, "Bruss", "Melese", "Bhavanasi", "Kuloe", "Izabel", "Beaume", "Liberopoulos").

BIGBOOK [www.bigbook.com], a yellow pages telephone directory. The query is a yellow pages category; $K = 6$ attributes are extracted: company name, street address, city, state, area code, and local phone number. Responses to 235 queries were collected; the queries were standard yellow pages categories (*e.g.*, "Automotive", "Business Services", "Communications", "Computers", "Entertainment and Hobbies", "Fashion and Personal Care").

COREL [corel.digitalriver.com], a searchable stock photography archive. They query is a series of keywords; $K = 3$ attributes are extracted: image URL, image name, and image category. Responses to 200 queries were collected; the queries were hand-selected from an English dictionary (*e.g.*"artichoke", "africa", "basket", "industry", "instruments", "israel", "japan", "juice").

ALTAVISTA [www.altavista.digital.com], a search engine. The query is a series of keywords; $K = 3$ attributes are extracted: URL, document title, and document summary. Responses to 300 queries were collected; the queries were randomly selected from an English dictionary (*e.g.*, "usurer", "farm", "budgetary", "wonderland", "peanut").

The OKRA resource was used to test and debug our implementation, and thus it is possible that we inadvertently tuned our system to this resource. To ensure impartiality, we therefore made no changes to the system when running it with the other resources.

We generated the labels for each example page by using a wrapper constructed

[3] The OKRA service was discontinued after this experiment was conducted.

by hand. These wrappers (one per resource) were used *only* to label the pages; the induction algorithm did not have access to them.

Figure 7.4 provides one measure of the performance of our system: the number of examples required for the induction algorithm to learn a wrapper that works perfectly on a suite of test problems. On each trial, we randomly split the collected examples into two halves: a training set, and a test set. We ran our system by providing it with the training set. Our methodology was to give the system just one training example, then two, then three, and so forth, until our system learned a wrapper that performed perfectly on the test set. Each such trial was repeated 30 times. Our results demonstrate that, for the sites we examined, only a handful of examples are needed to learn a high-quality wrapper.

Of course, these results assume a perfect recognizer library. Therefore, for four of the sites (OKRA, BIGBOOK, COREL and ALTAVISTA), we performed a more detailed analysis involving imperfect recognizers.

Specifically, after generating the correct labels, the labels were subjected to a mutation process, in order to simulate imperfect recognizers. We tested our system by varying which attributes were imperfectly recognized, as well as the rate at which these mistakes were made. The result was a set of trials, each corresponding to a different level of mistakes made in the recognition of different attributes.

Choosing the number of attributes to mutate. Specifically, to generate the trials, we first varied the number G of such imperfect recognizers from zero up to $K-1$. (Recall that our Corrob* algorithm requires that at least one attribute must be recognized perfectly; see Assumption 6.1.) For example, with BIGBOOK, we mutated between zero and five of the six attributes.

Choosing the attributes to mutate. Next, we enumerated the $\frac{K!}{G!(K-G)!}$ ways to select which G attributes to mutate from the K total. For example, when mutating two ($G = 2$) of BIGBOOK's attributes, there are $\frac{6!}{2!(6-2)!} = 15$ ways to choose two attributes out of six.

Assigning unsound and incomplete recognizers. Next, for a given set of G at-

resource	examples needed
OKRA	3.5
BIGBOOK	15.0
COREL	2.0
ALTAVISTA	2.0
survey site 1	4.5
survey site 3	2.2
survey site 4	2.2
survey site 5	2.0
survey site 8	2.0
survey site 10	2.1
survey site 12	2.0
survey site 13	2.0
survey site 14	4.8
survey site 15	2.0
survey site 19	2.0
survey site 20	2.0
survey site 22	2.0
survey site 23	2.0
survey site 25	2.0
survey site 27	2.0
survey site 30	3.1
average	3.0

Figure 7.4: *Average number of example pages needed to learn an* HLRT *wrapper that performs perfectly on a suite of test pages, for 21 actual Internet resources.*

tributes to be mutated, we enumerated all combinations of ways to introduce either false positives (simulating an unsound recognizer) or false negatives (simulating an incomplete recognizer). (Recall that our Corrob* algorithm can not handle recognizers that exhibit both false positives and false negatives; see Assumption 6.1.) For example, if we decide to mutate $G = 2$ attributes, and then select the URL and title attributes, then there are four possibilities: both have false positives, both have false negatives, title has false positives and URL has false negatives, and URL has false positives and title has false negatives.

Varying the mutation rate. Once the specific kinds of mutations are chosen, we then varied the *rate* τ at which these errors are introduced. We use the five rates $\tau \in \{0, 0.1, 0.2, 0.3, 0.4\}$. (Of course, $\tau = 0$ corresponds to a perfect recognizer.)

Performing the mutations. Finally, the selected attributes were mutated in the selected way, at the selected rate.

To introduce a false negative rate of τ, a fraction τ of the correct instances were simply discarded from the set of recognized instances. For example, with $\tau = 0.2$, if there are five correct instances, then we randomly select and discard one of them.

To introduce a false positive rate of τ, we introduced an additional $\frac{\tau}{1-\tau}W$ randomly constructed instances, where W is the number of correct instances of the selected attribute. These false positives were constructed by randomly selecting an integer $b \in [1, |P|]$, where $|P|$ is the length of the page whose label is being mutated. We then randomly select a second integer $l \in [\alpha-\beta, \alpha+\beta]$. Finally, we add the pair $\langle b, b+l \rangle$ into the set recognized instances. As desired, after adding these extra instances, a fraction τ of the entire collection are false positives. In the experiments we used $\alpha = 10$ and $\beta = 5$, so that the inserted instances have length between 5 and 15 characters, typical values for these resources.

One iteration of this five-step process corresponds to a single trial. Note that the number of trials is a function of K and thus there are different numbers of trials for the different resources in this experiment. Since randomness is introduced at several points, each trial was repeated several times. There are 257 trials for OKRA, and we ran each trial 10 times; there are 2657 trials for BIGBOOK, and we ran 5 iterations of each; and there are 73 trials for COREL and ALTAVISTA, and we ran 10 iterations of each.

As before, our methodology was to randomly split the collected examples into a training set and a test set, and then determine the minimum number of training examples required to learn a wrapper that performs perfectly on the test set.

We use this experimental setup to measure two statistics: the *number of examples* and the *CPU time* needed to perform perfectly on the test set, as a function of the level of recognizer errors (*i.e.*, the number of imperfectly recognized attributes, and the rate at which mistakes were made).

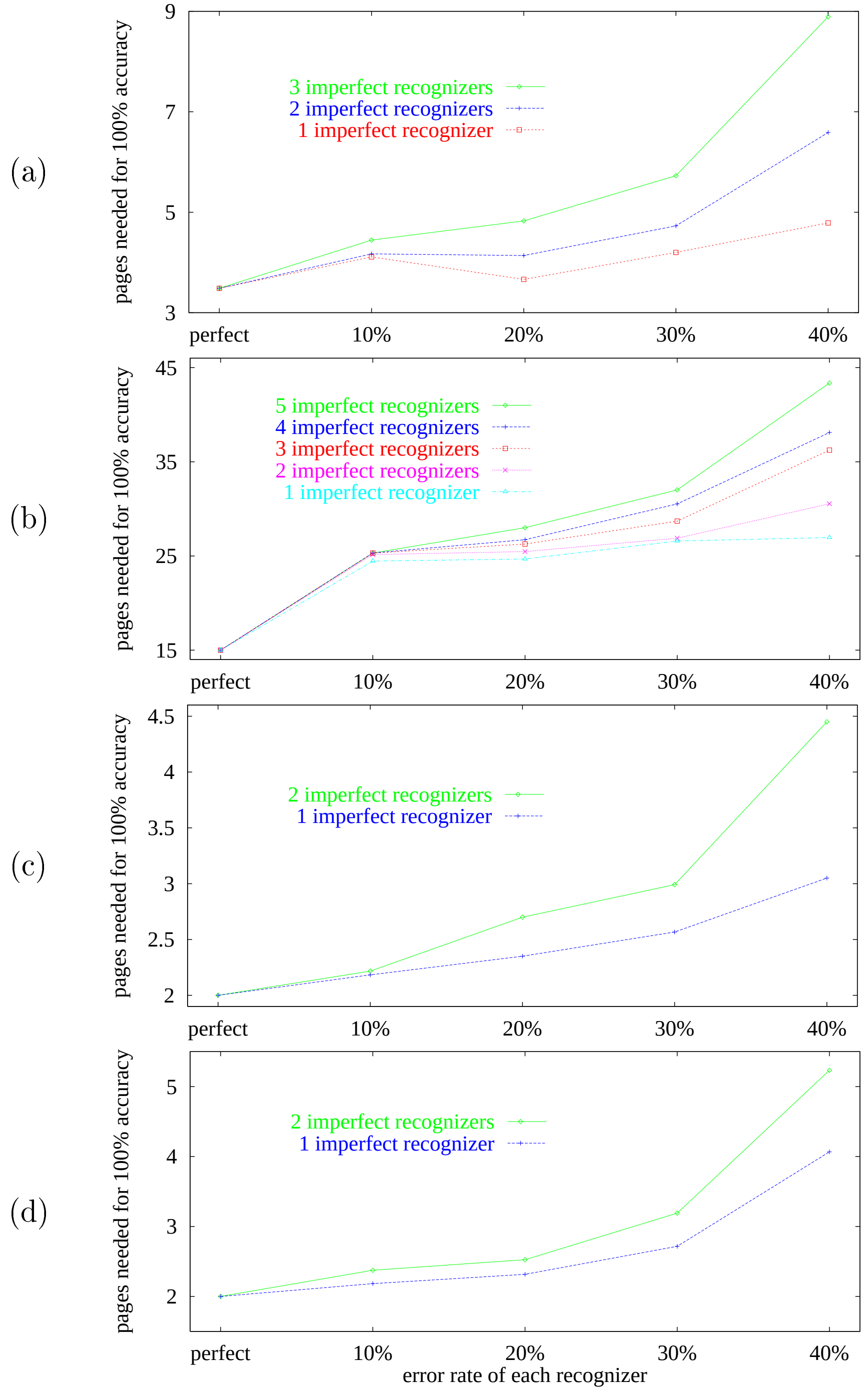


Figure 7.5: *Number of examples needed to learn a wrapper that performs perfectly on test set, as a function of the recognizer noise rate, for the (a)* OKRA*, (b)* BIGBOOK*, (c)* COREL *and (d)* ALTAVISTA *sites.*

Figure 7.5 shows our results for the first statistic, the number of examples needed to achieve perfect performance on the test set. Figure 7.5(a) shows our results for the OKRA resource; Figure 7.5(b), for BIGBOOK; Figure 7.5(c), for COREL; and Figure 7.5(d), for ALTAVISTA.

Within each graph, the different curves indicate different numbers of imperfect attribute recognizers (different values for G in the above discussion). For simplicity, each curve is the average across the various trials for the given value of G. The abscissa measures the noise rate (τ above) for each curve.

For example, in Figure 7.5(a), the top-most curve, marked "3 imperfect recognizers", represents all trials in which three of OKRA's attributes were recognized with a certain level of noise. The right-most point on this curve, marked "error rate of each recognizer 40%", indicates that in these trials the 3 imperfect recognizers made errors with noise rate $\tau = 0.4$. Notice that this top-right-most point represents 32 distinct trials: there are $\frac{4!}{3!(4-3)!}$ ways to choose three out of four attributes to mutate, times 2^3 ways to assign unsound or incomplete recognizers to the mutated attributes.

From Figure 7.5, we can see that our HLRT induction system requires between 2 and 44 example pages to learn a wrapper that performs perfectly on the test set, depending on the domain and the level of noise made by the recognizers. We conclude that wrapper induction requires a relatively modest sample of training examples to perform well.

Of course the total running time is at least as important as the required number of examples. Averaging across all the trials for each resource, our system performs as shown in Figure 7.6.[4] We conclude that our induction system does indeed run quite quickly—under ten seconds for three domains, and well under two minutes for the fourth.[5]

[4] All experiments were run on SPARC-10 and SGI Indy workstations; our system is implemented in Allegro Common Lisp.

[5] Let us emphasize that (in contrast to wrapper *execution*) wrapper *induction* is an off-line process—

Internet resource	total CPU time (sec.)	CPU time per example (sec.)
OKRA	5.02	1.57
BIGBOOK	83.5	4.49
COREL	1.28	0.56
ALTAVISTA	7.15	3.13

Figure 7.6: *Average time to learn a wrapper for four Internet resources.*

The reported CPU time is consumed by the two main algorithms developed in this thesis, $\mathsf{Generalize}_{\mathrm{HLRT}}$ (Chapter 4) and Corrob^* (Chapter 6). Of the two, Corrob^* is much faster: averaging across all resources and all conditions, Corrob^* takes only about 0.5% of the CPU time; the remaining 95.5% is consumed by $\mathsf{Generalize}_{\mathrm{HLRT}}$.

Why does BIGBOOK take so much longer than the other resources? Inspection of the query responses reveals that BIGBOOK's responses are much longer (about 25,000 characters per response, compared to about 10,000). As our complexity analysis shows (Theorem 4.7), our induction algorithm runs in time that is polynomial in the length of the examples. Thus the difference in time per example is explained by the difference in response length.

However, processing time per example does not completely explain the difference: BIGBOOK also requires many more examples than the other resources. For instance, Figure 7.5 shows that BIGBOOK requires about 15 examples with perfect recognizers, while the other resources require only 2–4 examples. The difficulty is due to advertisements. The heads of the BIGBOOK responses contain one of several different advertisements. To learn the head delimiter h, examples with different advertisements must be observed. But since there are relatively few advertisements, the chance of

perhaps fresh wrappers are learned each Sunday evening for an agent's on-line use during the week—and thus we consider even several minutes to be quite reasonable.

observing pages with the same advertisement is relatively high.

7.4 Evaluating the PAC model

We now turn to an evaluation of the PAC model. As discussed at the end of Section 4.6, our approach to evaluating the PAC model is enpirical rather than analytical. The PAC model *predicts* how many examples are required to learn a high-quality wrapper; in this section we compare this prediction with the numbers *measured* in the previous section.

We use the same methodology and resources as in Section 7.3. The PAC parameters were set rather loosely: we used the value $\epsilon = 0.1$ for the accuracy parameter, $\delta = 0.1$ for the reliability parameter. Our results are shown in Figure 7.7.

As Figure 7.7 shows, we found the PAC model predicts that between 251 and 1131 examples are needed to satisfy the PAC termination condition, depending on the domain and the level of noise made by the recognizers.

In constrast, earlier we saw that in fact only 2–44 examples are needed to learn a satisfactory wrapper. We conclude that our PAC model is too loose by one to two orders of magnitude. Moreover, note that the experiment in Section 7.3 involved learning wrappers that perform perfectly on large test suites, which (presumably) corresponds to tighter ϵ and δ values than the relatively loose parameters chosen here.

7.5 Verifying Assumption 4.1: Short page fragments

In Section 4.5 and throughout Chapter 5, we appealed to Assumption 4.1 as a basis for a heuristic-case analysis of the running time of the $\mathsf{Generalize}_{\mathcal{W}}$ function, for each wrapper class $\mathcal{W}$. In this section we empirically validate this assumption.

We begin by reviewing Assumption 4.1. Our $\mathsf{Generalize}_{\mathcal{W}}$ algorithms perform many string operations: searching for one string in another, and enumerating various

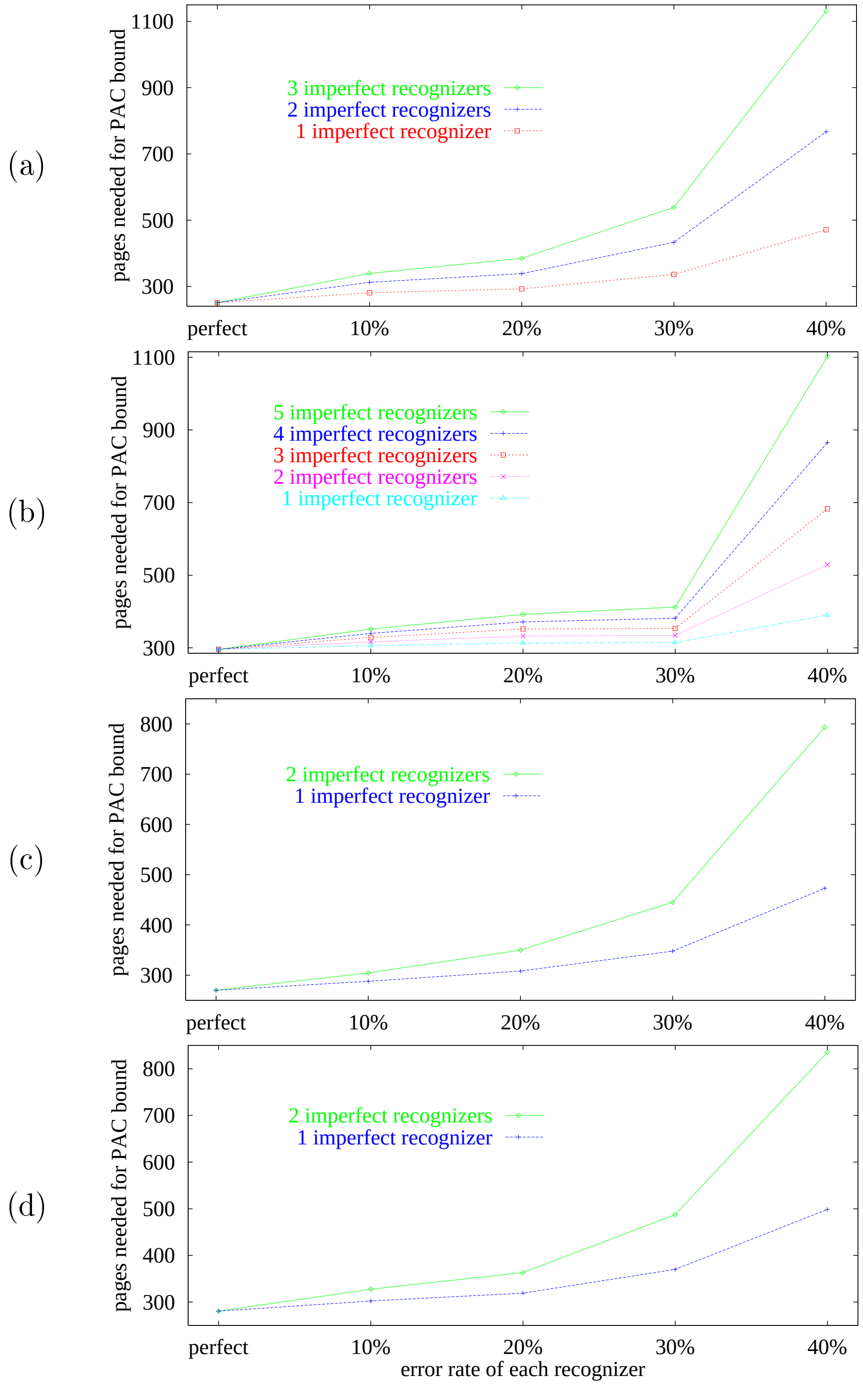


Figure 7.7: *Predicted number of examples needed to learn a wrapper that satisfies the PAC termination criterion, as a function of the recognizer noise rate, for the (a)* OKRA*, (b)* BIGBOOK*, (c)* COREL *and (d)* ALTAVISTA *sites.*

strings' substrings, suffixes, and prefixes. Our complexity analysis requires bounding the running time of these operations. To do so, we need to bound the length F of the strings over which these operations occur.

In our worst-case analysis, we bounded F by using the length $R = \min_n |P_n|$, the length of the shortest example page. Specifically, we simply used the bound $F = R$.

It is difficult to improve this bound using only analytical techniques. We therefore performed an heuristic-case analysis by asssuming that the relevant strings have lengths bounded by $F = \sqrt[3]{R}$ rather than $F = R$. In Section 4.5, we codified this approach as Assumption 4.1. We then used this assumption to derive tighter bounds on the running time of our functions $\mathsf{Generalize}_{\mathcal{W}}$ (*e.g.*, Theorem 4.7 in Section 4.5 for the HLRT wrapper class).

Of course the validity of these tighter bounds depends on the validity of Assumption 4.1. In order to verify Assumption 4.1, we simply compared the *actual* lengths with the *assumed* lengths.

Specifically, we examined the example pages collected for the four example resources described earlier in this chapter. We compared the length R of these examples pages with the actual length F of the $A_{m,k}$ and $S_{m,k}$ partition elements. (Note that these partition elements are exactly the strings discussed above whose lengths we want to bound.) Figure 7.8 shows a scatter-plot of the 121,868 $\langle F, R\rangle$ pairs collected from the example document, as well as the function $F = \sqrt[3]{R}$.

Since the data are so noisy, it is difficult to tell the extent to which Assumption 4.1 holds. We measured this support in three ways. First, we fit the data to a model of the form $F = \sqrt[\kappa]{R}$. We found that the value $\kappa = 3.1$ minimizes the least–squared-error between the model and the data. Figure 7.8 also shows the best-fit model $F = \sqrt[3.1]{R}$. Note that if $\kappa > 3$, then Assumption 4.1 is confirmed, because it overestimates the page fragment lengths.

As a second technique for demonstrating that the assumption holds, we measured the fraction of the data that lies below the curve $F = 4\sqrt[3]{R}$. We find that more

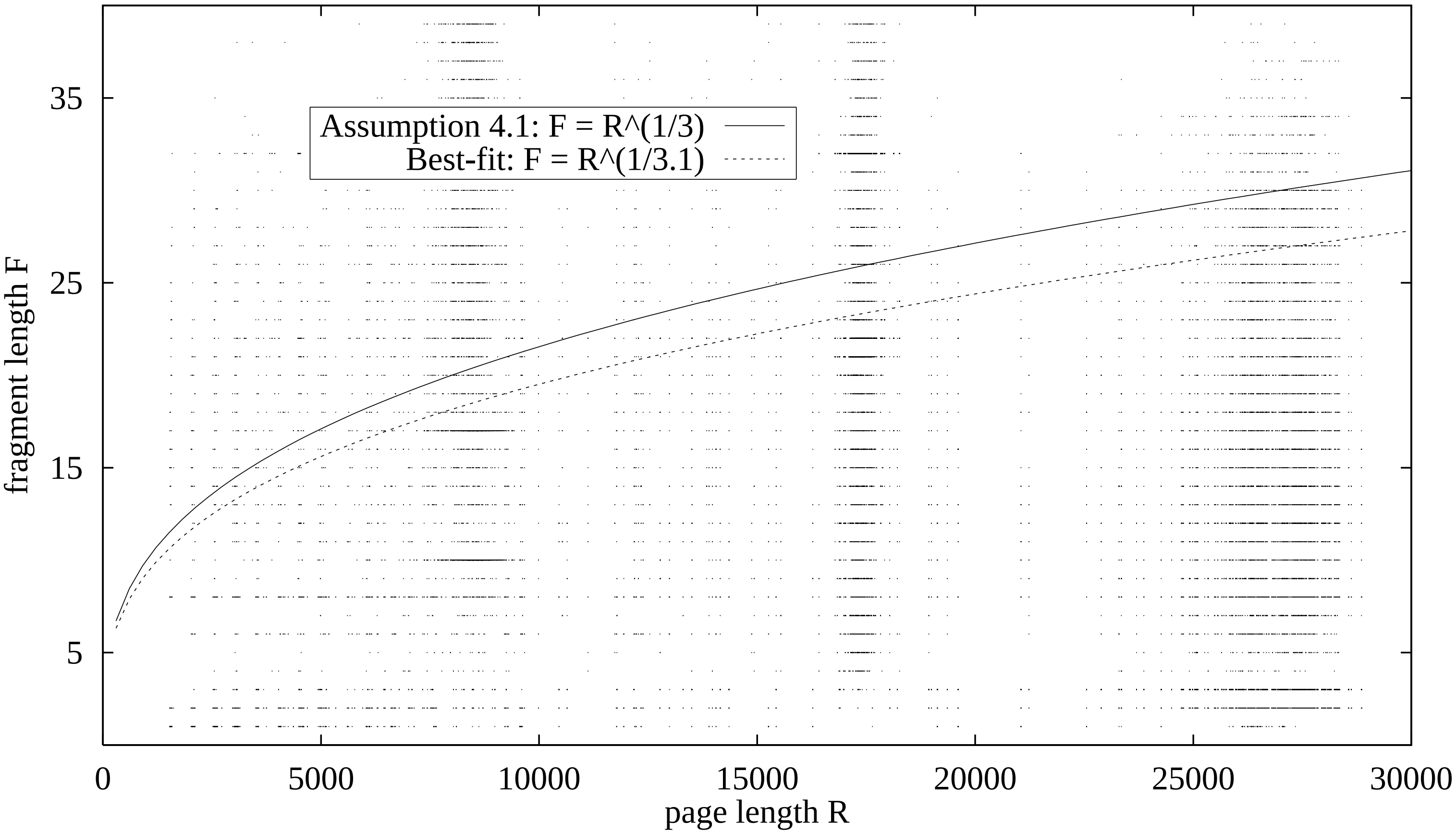


Figure 7.8: *A scatter-plot of the observed partition fragment lengths F* versus *page lengths R, as well as two models of this relationship: $F = \sqrt[3]{R}$ (the model demanded by Assumption 4.1), and $F = \sqrt[3.1]{R}$ (the best-fit model).*

than 80% of the data lie below this curve. Thus while the data contain points that violate Assumption 4.1, the majority of the data is within a small constant factor of assumption's predictions.

Finally, we randomly selected 50,000 of the $\langle F, R\rangle$ pairs, and measured the ratio $\frac{|F-\sqrt[3]{R}|}{R}$, which indicates the closeness of the predicated and observed lengths. Assumption 4.1 is validated to the extent that this ratio is zero. We find that the average value of this ratio is 0.7%.

Based on these three ways of comparing the predicted and observed page fragment lengths, we conclude that the data drawn from our actual Internet resources validate Assumption 4.1.

7.6 Verifying Assumption 4.2: Few attributes, plentiful data

In Chapter 4, we developed a PAC model for learning the HLRT wrapper class:

$$\Psi(K)\left(1-\frac{\epsilon}{\Psi(K)}\right)^{M_{\text{tot}}} + \Phi(R)\left(1-\frac{\epsilon}{2}\right)^{N} < \delta$$

(see Section 4.6 for details). Assumption 4.2 states that the sum on the left side of this inequality is dominated by its second term:

$$\Psi(K)\left(1-\frac{\epsilon}{\Psi(K)}\right)^{M_{\text{tot}}} \ll \Phi(R)\left(1-\frac{\epsilon}{2}\right)^{N}$$

In this section we empirically verify whether this assumption does in fact hold. Specifically, for each of the four resources introduced earlier in this chapter, we measured the ratio

$$\frac{\Psi(K)\left(1-\frac{\epsilon}{\Psi(K)}\right)^{M_{\text{tot}}}}{\Phi(R)\left(1-\frac{\epsilon}{2}\right)^{N}}. \tag{7.1}$$

(As before, for this experiment, we set $\epsilon = 0.1$.) Clearly, Assumption 4.2 is validated to the extent that this ratio is equal to zero.

Our experiment was designed as follows. For each of the four resources, we iterated the following process 5000 times. First, we selected a random integer N between 100

Internet resource	average ratio	maximum ratio
OKRA	7×10^{-7}	2×10^{-4}
BIGBOOK	2×10^{-7}	4×10^{-6}
COREL	$< 10^{-7}$	$< 10^{-7}$
ALTAVISTA	$< 10^{-7}$	$< 10^{-7}$

Figure 7.9: *The measured values of the ratio defined in Equation 7.1.*

and 400. We then randomly selected N examples pages,[6] and computed the ratio listed in Equation 7.1.

Figure 7.9 summarizes our results. For each resource, we report the average and maximum values of the ratio defined in Equation 7.1, over the 5000 trials. We conclude that Assumption 4.2 is strongly validated for the four resources we examined.

7.7 Measuring μ: The PAC model noise parameter (Equation 6.7)

Recall from Chapter 6 that if the recognizer library Δ contains imperfect recognizers, then the Corrob^* algorithm might output more than one label for a given page, because the labels might all be consistent with the evidence provided by the recognizers. Of course, only one such label is correct, and if a page is incorrectly labeled, then the induction system might output the wrong wrapper.

Without additional input, Corrob^* has no basis for preferring one label over another. As described in Chapter 6, our strategy for dealing with this problem is very simple: $\mathsf{Generalize}^{\text{noisy}}_{\text{HLRT}}$ tries to generalize from the noisy labels; we can only hope

[6] Note that Figure 7.7 indicates that, with perfect recognizer, the four resources require about 300 pages to satisfy the PAC termination criterion. Therefore, evaluating Assumption 4.2 for either very few, or very many, examples is unnecessary: in both case both the original PAC model and the approximation agree. Therefore, the most significant test for Assumption 4.2 involves N near 300.

that no consistent wrapper will exist, so that $\mathsf{Generalize}^{\text{noisy}}_{\text{HLRT}}$'s line 6.2(a) will try the next combination of labels. Since $\mathsf{Generalize}^{\text{noisy}}_{\text{HLRT}}$ tries all such combinations, eventually the correct combination of labels will be tried, and $\mathsf{Generalize}^{\text{noisy}}_{\text{HLRT}}$ will find and return the correct wrapper.

Unfortunately, as described in Chapter 6.5.2, this strategy ruins our PAC analysis, because there is always chance that we will inadvertently encounter a set of examples that are incorrectly labeled yet for which there exists a consistent wrapper. We extended our PAC model by introducing a noise rate μ, which quantifies this chance (Equation 6.6). We want μ to be small, because (as Equation 6.7 shows) as μ approaches zero, the PAC model requires fewer examples.

So far, we have given only an informal argument that μ is small. If our system is trying to learn how to extract the country from a page containing $\cdots$`<B>Congo</B>`$\cdots$, but the page's label extracts the string `B>Con` instead of `Congo`, then it is quite likely that no consistent wrapper will be found.

We now provide empirical evidence that μ is indeed very small. We repeated the following process 5000 times for each of the four information resources introduced earlier in this chapter. First, we randomly selected 15 example pages. We then mutated a small fraction of these pages' true labels. Specifically, we examined each pair $\langle b, e\rangle$. One percent of the time, we replaced this pair with a different pair, either $\langle b+\alpha, e\rangle$ or $\langle b, e+\alpha\rangle$. The choice of whether to change b or e is random. In each case α is randomly selected from the range $[-4, 4]$. The effect is that 1% of the $\langle b, e\rangle$ pairs are slightly "jiggled", in an attempt to simulate labels that are correct except for a few minor errors. We then passed these mutated labels to $\mathsf{Generalize}_{\text{HLRT}}$, which (as usual) tries to find a wrapper that is consistent with the examples.

Our results are as follows. Across all twenty thousand trials, we observed no cases for which there exists a consistent HLRT wrapper. Recall that the noise rate μ measures the chance of such a case, and so we conclude that for the Internet resources we examined, μ is extremely close to zero.

7.8 The WIEN application

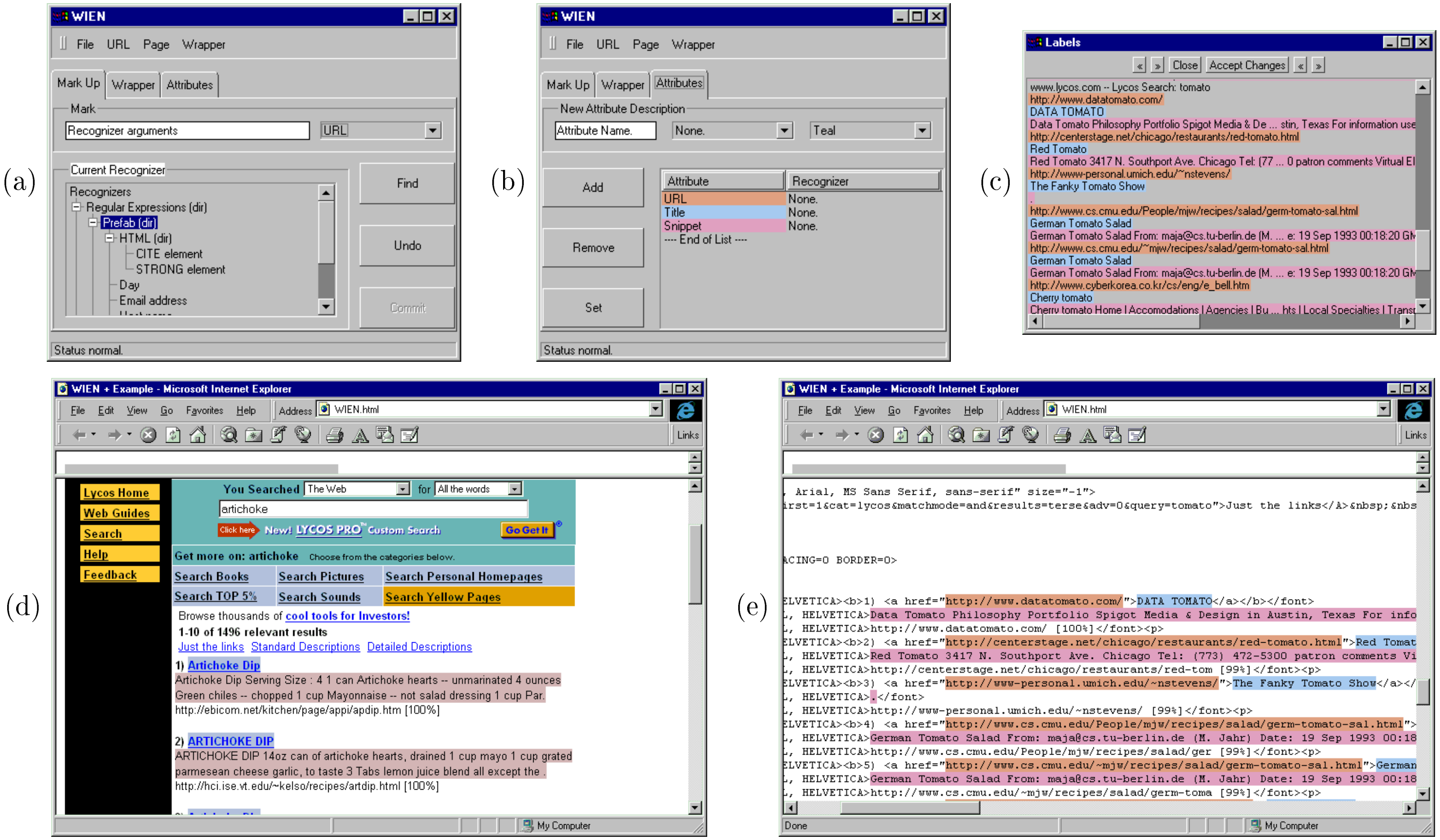
As an additional attempt to validate the techniques proposed in this thesis, we have developed WIEN, a *wrapper induction environment* application; see Figure 7.10.[7] Using a standard Internet browser, a user shows WIEN an example document (a page from LYCOS in this example). Then with a mouse the user indicates the fragments of the page to be extracted (*e.g.*, the document URL, title, and summary). WIEN then tries to learn a wrapper for the resource. When shown a second example, WIEN uses the learned wrapper to automatically label the new example. The user then corrects any mistakes, and WIEN generalizes from both examples. This process repeats until the user is satisfied.

In terms of the techniques developed in this thesis, WIEN provides a complete implementation of the $\mathsf{Generalize}_{\mathrm{HLRT}}$ algorithm, with the user playing the role of the example oracle $\mathsf{Oracle}_{\mathcal{T},\mathcal{D}}$.

The current version of WIEN does not implement the Corrob^* algorithm. However, as shown in Figure 7.10(a), we have provided an extensible facility for applying recognizers to the pages. When WIEN starts, it dynamically loads from a repository a library of recognizers. The user selects which recognizers are to be applied, and then manually corrects any mistakes. Thus while the user plays the role of the Corrob^* algorithm, its input recognizer library Δ is applied automatically.

In addition to this basic functionality, WIEN provides several facilities designed to simplify the process of automatic wrapper construction. Figure 7.10(b) shows WIEN's mechanism for editing the set of attributes to be extracted. Each attribute is assigned a color, which is used to highlight the extracted text in the browser window (7.10(d)). Since it is easy to make minor mistakes when marking up the pages, a facility is provided for reviewing and editing the extracted fragments (7.10(c)). Finally, since some extracted attributes might not appear when the page is rendered (*e.g.*, the URL

[7] WIEN is pronounced like "Vienna".


(a)
WIEN
File URL Page Wrapper
Mark Up Wrapper Attributes
Mark
Find
Undo
Commit
(b)
WIEN
File URL Page Wrapper
Mark Up Wrapper Attributes
New Attribute Description
Add
Remove
Set
(c)
Labels
« » Close Accept Changes « »
(d)
WIEN + Example - Microsoft Internet Explorer
File Edit View Go Favorites Help Address
Links
My Computer
(e)
WIEN + Example - Microsoft Internet Explorer
File Edit View Go Favorites Help Address
Links
My Computer

(a)

Recognizer arguments URL

Current Recognizer

Recognizers
- Regular Expressions (dir)
  - Prefab (dir)
    - HTML (dir)
      - CITE element
      - STRONG element
    - Day
    - Email address

Status normal.

(b)

Attribute Name. None. Teal

Attribute	Recognizer
URL	None.
Title	None.
Snippet	None.
---- End of List ----	

Status normal.

(c)

```
www.lycos.com -- Lycos Search: tomato
http://www.datatomato.com/
DATA TOMATO
Data Tomato Philosophy Portfolio Spigot Media & De ... stin, Texas For information use
http://centerstage.net/chicago/restaurants/red-tomato.html
Red Tomato
Red Tomato 3417 N. Southport Ave. Chicago Tel: (77 ... 0 patron comments Virtual El
http://www-personal.umich.edu/~nstevens/
The Fanky Tomato Show
.
http://www.cs.cmu.edu/People/mjw/recipes/salad/germ-tomato-sal.html
German Tomato Salad
German Tomato Salad From: maja@cs.tu-berlin.de (M. ... e: 19 Sep 1993 00:18:20 GM
http://www.cs.cmu.edu/~mjw/recipes/salad/germ-tomato-sal.html
German Tomato Salad
German Tomato Salad From: maja@cs.tu-berlin.de (M. ... e: 19 Sep 1993 00:18:20 GM
http://www.cyberkorea.co.kr/cs/eng/e_bell.htm
Cherry tomato
Cherry tomato Home | Accomodations | Agencies | Bu ... hts | Local Specialties | Trans
```

(d)

WIEN.html

Lycos Home
Web Guides
Search
Help
Feedback

You Searched The Web for All the words

artichoke

Click here New! LYCOS PRO™ Custom Search Go Get It

Get more on: artichoke Choose from the categories below.

Search Books	Search Pictures	Search Personal Homepages
Search TOP 5%	Search Sounds	Search Yellow Pages

Browse thousands of cool tools for Investors!

1-10 of 1496 relevant results

Just the links Standard Descriptions Detailed Descriptions

1) **Artichoke Dip**
Artichoke Dip Serving Size : 4 1 can Artichoke hearts -- unmarinated 4 ounces Green chiles -- chopped 1 cup Mayonnaise -- not salad dressing 1 cup Par.
http://ebicom.net/kitchen/page/appi/apdip.htm [100%]

2) **ARTICHOKE DIP**
ARTICHOKE DIP 14oz can of artichoke hearts, drained 1 cup mayo 1 cup grated parmesean cheese garlic, to taste 3 Tabs lemon juice blend all except the .
http://hci.ise.vt.edu/~kelso/recipes/artdip.html [100%]

(e)

WIEN.html

```
, Arial, MS Sans Serif, sans-serif" size="-1">
irst=1&cat=lycos&matchmode=and&results=terse&adv=0&query=tomato">Just the links</A>&nbsp;&nbs



ACING=0 BORDER=0>


ELVETICA><b>1) <a href="http://www.datatomato.com/">DATA TOMATO</a></b></font>
L, HELVETICA>Data Tomato Philosophy Portfolio Spigot Media & Design in Austin, Texas For info
L, HELVETICA>http://www.datatomato.com/ [100%]</font><p>
ELVETICA><b>2) <a href="http://centerstage.net/chicago/restaurants/red-tomato.html">Red Tomat
L, HELVETICA>Red Tomato 3417 N. Southport Ave. Chicago Tel: (773) 472-5300 patron comments Vi
L, HELVETICA>http://centerstage.net/chicago/restaurants/red-tom [99%]</font><p>
ELVETICA><b>3) <a href="http://www-personal.umich.edu/~nstevens/">The Fanky Tomato Show</a></
L, HELVETICA>.</font>
L, HELVETICA>http://www-personal.umich.edu/~nstevens/ [99%]</font><p>
ELVETICA><b>4) <a href="http://www.cs.cmu.edu/People/mjw/recipes/salad/germ-tomato-sal.html">
L, HELVETICA>German Tomato Salad From: maja@cs.tu-berlin.de (M. Jahr) Date: 19 Sep 1993 00:18
L, HELVETICA>http://www.cs.cmu.edu/People/mjw/recipes/salad/ger [99%]</font><p>
ELVETICA><b>5) <a href="http://www.cs.cmu.edu/~mjw/recipes/salad/germ-tomato-sal.html">German
L, HELVETICA>German Tomato Salad From: maja@cs.tu-berlin.de (M. Jahr) Date: 19 Sep 1993 00:18
L, HELVETICA>http://www.cs.cmu.edu/~mjw/recipes/salad/germ-toma [99%]</font><p>
```

Done

Figure 7.10: *The* WIEN *application being used to learn a wrapper for* LYCOS.

attribute), we have provided a facility for examining and marking up the raw HTML (7.10(e)).

Finally, we have extended WIEN in several ways not discussed elsewhere in this thesis. The most interesting extension is *aggressive learning*. If the user is willing to correct its mistakes, then he can invoke the learning algorithm before the example pages are completely labeled, and $\mathsf{Generalize}_{\mathrm{HLRT}}$ will do its best. Specifically, when invoked with incompletely labeled pages, $\mathsf{Generalize}_{\mathrm{HLRT}}$ might be able to find a wrapper that is consistent with all the examples. If such a wrapper can be found, it is then applied to the examples and any new extracted text fragments are identified. The user can then correct any mistakes, and invoke the regular learning algorithm on the resulting perfect labels. As Section 7.3 shows, relatively few examples are usually sufficient for effective learning, and so this aggressive learning mechanism considerably simplifies the wrapper construction process.

We have not conducted user studies with WIEN, although anecdotally it appears to be a useful tool for helping to automate the wrapper construction process.

Chapter 8

RELATED WORK

8.1 Introduction

Our approach to automatic wrapper construction draws on ideas from many different research areas. In this chapter we review this related literature by considering three kinds of work. First, we describe work related to the *motivation* and background of this thesis (Section 8.2). Second, we describe several projects that are concerned with similar *applications* (Section 8.3). Finally, we describe work that is related at a *formal* or theoretical level (Section 8.4).

8.2 Motivation

The systems that motivate our focus on wrapper induction are mainly concerned with the integration of heterogeneous information resources (Section 8.2.1). In addition, research on supporting legacy systems is also relevant (Section 8.2.2), as is work on the development of standard information exchange protocols (Section 8.2.3)

8.2.1 Software agents and heterogeneous information sources

As described in Section 1.1, our concern with wrapper induction can be traced to the pressing need for systems that interact with information resources that were designed to be used by *people* rather than *machines*. The hope is that such capabilities will relieve users of the tedium of directly using the growing plethora of on-line resources.

There are two distinct sub-communities working in this area. The first involves artificial intelligence researchers interested in *software agents*; see [Etzioni et al. 94, Wooldridge & Jennings 95, Bradshaw 97] for surveys. The second consists of database and information systems researchers working on the *integration of heterogeneous databases*; see [Gupta 89] for a survey.

We were strongly influenced by the University of Washington "softbot" projects [Etzioni et al. 93, Etzioni 93, Etzioni & Weld 94, Perkowitz & Etzioni 95, Selberg & Etzioni 95, Etzioni 96a, Kwok & Weld 96, Selberg & Etzioni 97, Doorenbos et al. 97, Friedman & Weld 97, Shakes et al. 97]. Related projects at other institutions include CARNOT [Collet et al. 91], DISCO [Florescu et al. 95], GARLIC [Carey et al. 95], HERMES [Adali et al. 96], the Information Manifold [Levy et al. 96], SIMS [Arens et al. 96], TSIMMIS [Chawathe et al. 94], FUSION [Smeaton & Crimmins 97], BargainFinder [Krulwich 96], and the Knowledge Broker [Andreoli et al. 96, Chidlovskii et al. 97].

While these systems are primarily research prototypes, there is substantial commercial interest in software agents and heterogeneous database integration products. Examples include Jango [www.jango.com], Junglee [www.junglee.com], AlphaCONNECT [www.alphamicro.com], BidFind [www.vsn.net/af], LiveAgent [www.agentsoft.com], Computer ESP [oracle.uvision.com/shop], the Shopping Explorer [shoppingexplorer.com], and Texis [thunderstone.com].

The details of these projects vary widely, but all share a common need for a layer of wrappers between the integration system and the information resources it accesses. For example, the Ahoy! system [Shakes et al. 97] searches for people's home pages by querying email address locator services, search engines, *etc.*, while Junglee [www.-junglee.com] integrates across multiple sources of apartment or job listings. Currently, each relies on hand-crafted wrappers to interact with the resources it needs.

Finally, let us mention that wrapper construction touches on subtle issues related to copyright protection in the information age. Does the owner of an on-line newspaper or magazine have the right to dictate that its pages can not be automatically parsed and re-assembled (*e.g.*, to remove advertisements) without permission?

Clearly, these so-called "digital law" issues are well beyond the scope of this thesis; see [infolawalert.com] for further information.

8.2.2 Legacy systems

While we are primarily motivated by the task of integrating heterogeneous information resources, our work is also related to the management of so-called *legacy systems* [Aiken 95, Brodie & Stonebraker 95]. For various economic and engineering reasons, many information repositories are buried under layers of outdated, complex or poorly-documented code.

Of particular relevance are legacy databases. One common strategy is to encapsulate the entire legacy database in a software layer that provides a standard interface. There is substantial effort are building such wrappers; see, for example, [Shklar et al. 94, Shklar et al. 95, Roth & Schwartz 97].

It remains an open question whether the techniques we have developed in this thesis can be applied to legacy systems. Certainly, few of these systems use HTML formatting. But we conjecture that the formatting conventions used do exhibit sufficient regularity that our techniques can be applied. For example, a system was recently developed to automatically execute queries against the University of Washington legacy "telnet" interface to the library catalog [Draper 97]. The system uses VT-100 escape sequences to format the requested content. Like the wrapper classes described in this thesis, the system uses constant strings such as `ESC[2j` or `Title:` to identify and extract the relevant content.

8.2.3 Standards

Finally, in Section 1.1.1 we described one objection to the very idea of wrappers: why not simply adopt standard protocols to facilitate communication between the information providers and consumers? A wide variety of such standards have been developed for various purposes; examples include CORBA [www.omg.org], ODBC [www.microsoft.com/data/odbc], XML [www.w3.org/TR/WD-xml], KIF [logic.stanford.edu/kif], Z39.50, [lcweb.loc.gov/z3950], WIDL [www.webmethods.com], SHOE [Luke et al. 97] and KQML [Finin et al. 94].

The main problem with all such standards is that they must be enforced. Unfortunately, despite the clear benefits, there are many reasons—economic, technical, and social—why adherance is not always universal. For example, on-line businesses might prefer to be visited manually rather than mechanically, and refusing to adhere to a standard is one way to discourage automatic browsing. Search engines such as Yahoo!, for example, are in the business of delivering users to advertisers, *not* servicing queries as an end in itself. Similarly, a retail store might not want to simplify the process of automatically comparing prices between vendors. And of course the cost of re-engineering existing resources might be prohibitive.

As with the earlier discussion of digital law, these issues are complex and beyond the scope of this thesis. The bottom line, though, is that certainly automatic wrapper construction will become less compelling if standards become entrenched.

8.3 Applications

We now turn to research that is focused on applications that are similar to our work on wrapper construction and information extraction.

8.3.1 Systems that learn wrappers

We know of three systems that are involved with learning wrappers: ILA, SHOPBOT, and ARIADNE.

ILA. The ILA system [Perkowitz & Etzioni 95, Perkowitz et al. 97] was proposed as an instance of the idea of a softbot learning how to use its tools. ILA takes as input a set of pre-parsed query responses; its task is to determine the *semantics* of the extracted text fragments. For example, when learning about the country/code resource, ILA is given the information content

$$\{\langle \texttt{Congo}, \texttt{242}\rangle,\ \langle \texttt{Egypt}, \texttt{20}\rangle,\ \langle \texttt{Belize}, \texttt{501}\rangle,\ \langle \texttt{Spain}, \texttt{34}\rangle\}$$

(Note that ILA would *not* directly manipulate the raw HTML (*e.g.* Figure 2.1(c)).) ILA then compares these values with a collection of facts such as:

$$\begin{array}{lll}
\text{name}(c_1) = \texttt{Congo} & \text{country-code}(c_1) = \texttt{242} & \text{capital}(c_1) = \texttt{Brazzaville} \\
\text{name}(c_2) = \texttt{Ireland} & \text{country-code}(c_2) = \texttt{353} & \text{capital}(c_2) = \texttt{Dublin} \\
\text{name}(c_3) = \texttt{Spain} & \text{country-code}(c_3) = \texttt{34} & \text{capital}(c_3) = \texttt{Madrid}
\end{array}$$

On the basis of this background knowledge, ILA hypothesizes that the first attribute is the name of the country, while the second is its telephone country code (rather than, for example, its capital city). Thus ILA has learned that the country/code resource returns pairs of the form $\langle \text{name}(c),\ \text{country-code}(c)\rangle$.

This example is very simple. For more complicated resources, ILA searches the space of functional relationships between the attribute values. For example, ILA might discover that some resource returns pairs consisting of a country and the address of the mayor of its capital city: $\langle \text{name}(c),\ \text{address}(\text{mayor}(\text{capital}(c)))\rangle$. Moreover, if necessary ILA asks focused queries to disambiguate the observed examples.

ILA and our wrapper construction techniques thus provide complementary functionality. ILA requires that the input examples be pre-parsed, but uses knowledge-based techniques for determining a resource's semantics. In contrast, our system uses a relatively primitive model of semantics (recognizers and the corroboration process), but learns how to parse documents. Clearly, integrating the two approaches is an interesting direction for future work.

SHOPBOT. A second system that learns wrappers is SHOPBOT [Doorenbos et al. 97, Perkowitz et al. 97]. In many ways, SHOPBOT is much more ambitious than our work.

As a full-fledged autonomous shopping agent, SHOPBOT not only learns how to extract a vendor's content, but also how to query the vendors, and how to identify unusual conditions (*e.g.*, a vendor not stocking some particular product). In this comparison, we focus exclusively on SHOPBOT's techniques for learning to parse a vendor's query responses.

SHOPBOT operates by looking for patterns in the HTML source of the example document. SHOPBOT first partitions the example pages into a sequence of logical *records*; the system assumes that these records are separated by visually salient HTML constructs such as <HR>, <LI>, or <BR>. Each record is then abstracted by removing non-HTML characters, generating a *signature* for each record. For example, when examining the country/code response in Figure 2.1(c), SHOPBOT would generate the following six signatures:

```
<HTML><TITLE>text</TITLE>⇓<BODY><B>text</B><P>
<B>text</B> <I>text</I><BR>
<B>text</B> <I>text</I><BR>
<B>text</B> <I>text</I><BR>
<B>text</B> <I>text</I><BR>
<HR><B>text</B></BODY></HTML>
```

SHOPBOT then ranks these signatures by the fraction of the pages each accounts for. In the example, the first and last signatures each account for one-sixth of the page, while the middle four identical signatures together account for four-sixths of the page. On this basis, SHOPBOT decides to ignore the parts of the page corresponding to the first and last signatures, and extracts from the country/code resource only those records that match the signature <B>*text*</B> <I>*text*</I>
.

SHOPBOT also uses domain-specific heuristics to rank these signatures. For example, in shopping tasks, the presence of a price suggests that a signature is correct. Moreover, these heuristics are used to extract some information from each record. For example, SHOPBOT attempts to extract prices from each record, so that its results can be sorted.

At this point we can compare SHOPBOT and our wrapper induction system. SHOPBOT uses HTML-specific heuristics to identify records. In contrast, our system learns to exploit—but does not depend on—HTML or any other particular formatting convention. Moreover, although SHOPBOT does partially extract the content of the individual records using domain-specific heuristics, it does not learn to fully parse each record. Thus SHOPBOT uses wrappers from a class that we might call HOCT: SHOPBOT learns to ignore pages' heads and tails and extract each tuple as a whole, but it does not learn to extract the individual attributes within each tuple.

ARIADNE. Finally, ARIADNE is a semi-automatic system for constructing wrappers [Ashish & Knoblock 97a, Ashish & Knoblock 97b]. ARIADNE is targeted at hierarchically structured documents, similar to those we discussed in Section 5.3.

ARIADNE employs powerful (though HTML- and domain-specific) heuristics for guessing the structure of pages. For instance, relative font size is a usually good clue for determining when an attribute ends and subordinate attributes begin. For example, given the following page:

```
Introduction to CMOS Circuits
<FONT SIZE=-1>CMOS Logic</FONT>
<FONT SIZE=-2>The Inverter</FONT>
MOS Transistor Theory
```

ARIADNE would hypothesize that `Introduction` $\cdots$ `Circuits` and `MOS` $\cdots$ `Theory` are instances of the highest-level headings, while `CMOS Logic` is a subheading and `The Inverter` is a sub-subheading. In addition to font sizes, ARIADNE attends to the use of bold or italics fonts, sequences of alphanumeric characters ending with a colon (*e.g.*, `Title:`), relative indentation, and several other heuristics.

Once the hierarchically nested structure of such a document is determined, ARIADNE generates a grammar consistent with the observed structure. Finally, a parser is generated that accepts "sentences" (*i.e.*, pages) in the learned grammar. Of course since the heuristics used to guide these processes are imperfect, the user must correct

ARIADNE's guesses. Ashish and Knoblock report that only a handful of corrections are needed in practice.

ARIADNE's grammars are similar to the N-LR wrappers defined in Section 5.3. The difference is that, in N-LR the ℓ_k and r_k delimiters must be constant strings, while they can be regular expressions in ARIADNE's wrappers. (Note, though, that ARIADNE does not learn these regular expressions, but rather imports them from the heuristics used to find the page's structure.)

8.3.2 Information extraction

At the highest level, this thesis is concerned with *information extraction* (IE). This field has a rich literature; see [Hobbs 92, Cowie & Lehnert 96] for surveys. Although similar in spirit—traditional IE is the task of identifying literal fragments of an input text that instantiate some relation or concept—our use of the phrase "information extraction" differs in five ways.

The first difference concerns our focus on extra-linguistic regularity to guide extraction. In contrast, with its roots in natural language processing (NLP), much other IE work is designed to exploit the rich linguistic structure of the sources documents. While such approaches are useful in many applications, we have found that the Internet resources which motivate us often do not exhibit this linguistic structure.

A second difference is that many knowledge-intensive approaches to IE are slow and thus best suited to off-line extraction, while a software agent's wrappers must execute on-line and therefore quickly.

A third difference is that we take a rather rigid approach to information extraction, demanding that the text to be extracted is reliably delimited by constant strings (the r_k and ℓ_k in our various wrapper classes). In contrast, NLP-based IE systems begin by segmenting the text, in effect inserting invisible delimiters around noun phrases, direct objects, *etc.* Our approach could in principle operate on unstructured text once it has been annotated with such syntactic delimiters; *e.g.*, adding `<NP>` and `</NP>`

around noun phrases, <DOBJ> and </DOBJ> around the direct objects, *etc.*.

A fourth difference is that most IE systems employ rules that extract relevant text fragments. A separate post-processing phase is usually employed to group these fragments together into a coherent summarization of the document. For instance, this post-processing phase is responsible for handling anaphora (*e.g.*, resolving pronoun referents) or merging fragments that were extracted twice by different rules. In contrast, our approach to information extraction is to identify the entire content of the page at once. Of course, this approach is successful only because the documents in which we are interested have certain kinds of regular structure.

Finally, given our goal of eliminating the engineering bottleneck caused by constructing wrappers by hand, we are interested in automatic learning techniques. In contrast, many IE systems are hand-crafted. The two notable exceptions are AUTOSLOG and CRYSTAL.

AUTOSLOG [Riloff 93] learns information extraction rules. The system uses heuristics to identify specific words that trigger extraction. For example, when learning to extract medical symptoms *not* experienced by patients, AUTOSLOG would learn that the word "denies..." indicates such symptoms, based on example sentences such as "The patient denies any episodes of nausea." While there are many important differences, it is apparent that this learned rule is similar to our wrappers in that a specific literal string is used to delimit the desired text.

Like AUTOSLOG, CRYSTAL [Soderland et al. 95, Soderland 97b] and its descendant WEBFOOT [Soderland 97c] learn information extraction rules. CRYSTAL takes as input a set of labeled example documents and a set of features describing these documents. As originally envisioned, CRYSTAL used linguistic features such as part-of-speech tags, and it learned rules that are triggered by these linguistic features. WEBFOOT extends CRYSTAL to handle non-linguistic Internet-based documents. WEBFOOT provides the CRYSTAL rule-learning algorithm with a set of features that are useful for information extraction from Internet-based (rather than free natural language) documents.

Specifically, a set of HTML-specific heuristics are used to identify the text fragments to be extracted; the heuristics are similar to those used by SHOPBOT and ARIADNE.

Similar ideas are explored in [Freitag 96, Freitag 97]. Using as a test domain relatively unstructured departmental talk announcements, Freitag demonstrates that different machine learning techniques can be combined to improve the precision with which various information extraction tasks can be performed.

These three systems—AUTOSLOG, CRYSTAL/WEBFOOT, and Freitag's work—suggest promising directions for future research. Drawing on techniques from both NLP and machine learning, they point to an integration of NLP-based techniques and our extra-linguistic techniques; see Section 9.2 for details.

8.3.3 Recognizers

Chapter 6 describes techniques for automating the process of labeling the example pages. Central to these techniques is a library of *recognizers*, domain-specific procedures for identifying instances of particular attributes.

Recognizers for specific attributes have received much attention in the text processing communities. For example, the Sixth Message Understanding Conference's "Named Entity" task [ARPA 95, ftp.muc.saic.com/pub/MUC/MUC6-guidelines/ne-task-def.v2.1.ps.Z] involves identifying particular kinds of information such as people and company names, dates, locations, and so forth.

Certain highly valuable attributes such as company and people's names have received substantial attention [Rau 91, Borgman & Siegfried 92, Paik et al. 93, Hayes 94]. This research has matured to the point that high-quality commercial name recognizers are now available—examples include the Carnegie Group's "NameFinder" system [www.cgi.com], and "SSA-NAME" [www.searchsoftware.com]. Our recognizers are also similar to Apple Computer's *Data Detectors* [applescript.apple.com/data_detectors].

Others have taken machine learning approaches to constructing recognizers for particular kinds of attributes. The WIL system [Goan et al. 96] uses novel grammar induction techniques to learn regular expressions from examples of the attribute's values; they demonstrate that their techniques are effective for learning attributes such as telephone numbers and US Library of Congress Call Numbers. [Freitag 96] presents similar results for the domain of academic talk announcements.

Finally, the "field matching" problem [Monge & Elkan 96] is relevant to building recognizers. Field matching involves determining whether two character strings, such as "*Dept. Comput. Sci. & Eng.*" and "*Department of Computer Science and Engineering*", do in fact designate the same entity. Monge and Elkan propose heuristics that are effective at solving the field matching problem in the domain of aligning academic department names and addresses. In Section 6.7, we suggested that one technique for building recognizers is to exploit existing indices—*e.g.*, constructing a company name recognizer from the Fortune 500 list. Such recognizers will probably perform poorly unless they solve the field matching problem.

8.3.4 Document analysis

Our approach to information extraction exploits the structure of the information resource's query responses. For example, the LR, HLRT, OCLR, and HOCLRT wrapper classes exploit the fact that the page is formatted with a tabular layout, while the N-LR and N-HLRT classes assume a hierarchically nested layout.

There is a wide variety of research concerned with recovering a document's structure. Of particular relevance to our work is the recovery of structured information such as tables of tables-of-contents. [Douglas et al. 95, Douglas & Hurst 96] discuss techniques for identifying the tabular structured in plain text documents. [Green & Krishnamoorthy 95] solve a similar problem, except that their system takes as input scanned images of documents.

More ambitiously, [Rus & Subramanian 97] provide a theoretical characterization of *information capture and access*, a novel approach to the development of systems that integrate heterogeneous information sources. The idea is to formalize the notion of a document *segmenter*, which identifies possibly-relevant fragments of the document. These candidates are then examined by *structure detectors*, which look for patterns among the segments. This work is interesting because many different kinds of heuristic techniques for identifying document structure can be modeled using their formalism. For example, when trying to identify a document's tabular structure, a segmenter might look for rectangular areas of white-space, and then a structure detector would try to locate repetitive arrangements of these regions that indicate a table.

Rus and Subramanian demonstrate that their techniques effectively identify the tabular structure of plain text documents such as retail product catalogs. They also demonstrate techniques for automatically extracting and combining information from heterogeneous sources. For example, they have developed an agent that scans various sources of stock price listings to generate graphs of performance over time. Unfortunately, this system relies on hand-coded heuristics to help it determine the *semantics* of the extracted information. Integrating these techniques with ILA [Perkowitz & Etzioni 95, Perkowitz et al. 97] would be an interesting direction for future work.

8.4 Formal issues

So far, we have described the related projects at a relatively shallow level of detail. The reason is simply that while our wrapper induction work is *motivated by* or *addresses similar issues as* other work, the nuts-and-bolts technical details are actually very different. However, there are two research areas which call for a more detailed comparison. First, we first briefly describe the connection between wrapper induction and grammar induction learning (Section 8.4.1). Second, we describe the relationship between our PAC model and others in the literature (Section 8.4.2).

8.4.1 Grammar induction

For each wrapper class $\mathcal{W}$, the Exec$\mathcal{W}$ procedure uses a finite amount of state to scan and parse the input page, augmented with additional book-keeping state for keeping track of the extracted information. Although unbounded, this book-keeping state is entirely unrelated to the state used for parsing, and thus our wrappers are formally equivalent to regular grammars. It is thus natural to compare our inductive learning algorithms with the rich literature in regular grammar induction; see for example [Biermann & Feldman 72, Gold 78, Angluin & Smith 83, Angluin 87].

We did not employ off-the-shelf grammar induction algorithms. To understand why, recall the purpose of wrappers: they are used for *parsing*, not simply classification. That is, our wrappers can not simply examine a query response and confirm that it came from a particular information resource. Rather, a specific sort of examination must occur; namely, one that involves scanning the page in such a way as to identify the text fragments to be extracted.

Therefore, we require that the finite-state automaton to which the learned grammar corresponds have a specific state topology. Efficient induction algorithms have been developed for several classes of regular grammars (*e.g.*, *reversible* [Angluin 82] and *strictly regular* [Tanida & Yokomori 92] grammars). The difficulty is simply that we do not know of any such results that deliver the particular state topology we require. Therefore, we have developed novel induction algorithms targeted specifically at our wrapper classes.

8.4.2 PAC model

Our PAC model (Equation 4.5 in Section 4.6) appears to be more complicated than most results in the literature.

The reason for this complexity is simply that we are learning relatively complicated hypotheses. Our proofs rely on elaborate constructions, such as our use of a disjunction of $2K$ terms to capture the ways that a wrapper's error can exceed ϵ (pages 217–219), or our use of an interval $[L, U]$ to represent the set of proper suffixes common to a given set of strings (pages 219–224).

However, once these constructions are in place, we make use of well-known proof techniques. Our proof of Lemma B.1's parts (1–2) is a simplified version of the well-known proof that axis-aligned rectangles are PAC-learnable [Kearns & Vazirani 94, pp 1–6], and our proof of part Lemma B.1 part (3) is so basic as to be published in AI textbooks (*e.g.*, [Russell & Norvig 95, pp 553–5]).

However, there are two areas of related work which must be addressed: a comparison of our results with the use of the Vapnik-Chervónenkis (VC) dimension, and our model of label noise.

The VC dimension. The *VC dimension* of a hypothesis class is a combinatorial measure of the inherent difficulty of learning hypotheses in the class [Vapnik & Chervónenkis 71]. The VC dimension of hypothesis class $\mathcal{H}$ is defined as the cardinality of the largest set $\mathcal{I}$ of instances for which there exists hypotheses in $\mathcal{H}$ that can classify $\mathcal{I}$'s members in all $2^{|\mathcal{I}|}$ possible ways.

For example, suppose we treat the intervals of the real number line as a hypothesis class, where an interval classifies as true those points enclosed in the interval. No matter how we pick any two real numbers, we can always find an interval that: contains both points, contains neither point, contains one point but not the other, and *vice versa*. Therefore, the VC dimension of the real interval hypothesis space is at least two. However, this property does not hold for *any* set of three real numbers: there is no interval that includes the two extreme points yet excludes the middle point. Therefore, the VC dimension of the real interval hypothesis space is exactly two.

The VC dimension is useful because PAC bounds for hypothesis classes with finite VC dimension have been developed [Haussler 88, Blumer et al. 89].

Analyzing the VC dimension of the HLRT wrapper class is somewhat complicated. We can easily analyze the VC dimension of the r_k and ℓ_k ($k > 1$) delimiters in isolation. Since we reduce the problem of learning the r_k and the ℓ_k (for $k > 1$) to the problem of learning an interval over the integers (see the proof of Lemma B.1 parts (1–2) is Section B.4) we know that the VC dimension of each of these $2K - 1$ components is two. However, what is the VC dimension of the "h, t, ℓ_1" subspace of the HLRT class? We conjecture that this VC dimension is infinite: since pages and the HLRT delimiters are all strings of unbounded length, there are essentially an unbounded number of degrees of freedom in the system. If this conjecture holds, then the bounds reported in [Haussler 88, Blumer et al. 89] are inapplicable. In summary, we speculate that the PAC bounds we report for HLRT are tighter than obtainable using VC-theoretic analysis.

Label noise. Finally, let us elaborate on the model of noise assumed in our extensions of the PAC model to handle corroboration (Section 6.5.2).

Recall that we use an extremely simple model of noise: Equation 6.6 defines the parameter μ, which measures the "danger" associated with labels' noise. The $\mathsf{Generalize}^{\text{noisy}}_{\text{HLRT}}$ algorithm simply assumes that wrappers can be found only for pages that are labeled correctly; μ measures how risky this assumption is.

Note that μ does *not* measure the chance that any particular label is wrong, that any particular set of labels are wrong, *etc.* Instead, μ plays the following role. The $\mathsf{Generalize}^{\text{noisy}}_{\text{HLRT}}$ algorithm (Figure 6.2) assumes that a consistent wrapper exists only for correctly labeled pages; μ is a measure of how likely $\mathsf{Generalize}^{\text{noisy}}_{\text{HLRT}}$ is to make a mistake by following this procedure. For example, a recognizers library Δ might make many mistakes, so that nearly all the labels returned by Corrob^* (and therefore tried by $\mathsf{Generalize}^{\text{noisy}}_{\text{HLRT}}$) contain errors. Nevertheless, μ might be close to zero, if the information resource under consideration is structured so that incorrect labels can be discovered by the $\mathsf{Generalize}_{\text{HLRT}}$ algorithm.

This approach to handling noise appears to be novel. The PAC literature (*e.g.* [Angluin & Laird 88, Kearns 93, Decatur & Gennaro 95]) covers a wide variety of noise models and strategies for coping with this noise.

The basic idea is that when the learning algorithm asks the $\mathsf{Oracle}_{\mathcal{T},\mathcal{D}}$ for any example, the oracle returns the correct example $\langle I, \mathcal{T}(I)\rangle$ with probability $1-\nu$, and returns a defective example $\langle I, L'\rangle$ with probability ν, where L' is a mutated version of $\mathcal{T}(I)$ (*e.g.*, if the labels are binary, then $L' = \neg\mathcal{T}(I)$), and $0 \leq \nu \leq 1$ is the chance that the oracle is wrong. Under these circumstances, reliable learning is a matter of obtaining enough examples (how many depends on ν) to compensate for the fact that any particular label might be wrong.

In summary, standard approaches to learning from noise examples involve extra sampling, in the hope that the noise will wash out. In contrast, we assume an oracle that provides not one but several labels for each example, and we use self-correction mechanism to verify that the observed labels are correct. Our experiments in Section 7.7 confirm that this self-correction mechanism is highly reliable (*i.e.* $\mu \approx 0$) for the Internet resources we have examined.

Chapter 9

FUTURE WORK AND CONCLUSIONS

9.1 Thesis summary

Let us review this thesis by discussing our three main contributions.

Contribution 1: Automatic wrapper construction as inductive learning. We began with a description of the kinds of information extraction tasks in which we are interested (Chapter 2): an *information resource* responds to *queries* with a *response* (*e.g.*, an HTML page) containing the *content* we are looking for. We focus on pages that are *semi-structured*, in that while they might contain advertisements and other extraneous information, the content itself obeys rigid (though arbitrary and unknown) formatting conventions. Specifically, we focus on resources whose content is formatted in either a *tabular* or *hierarchically nested* manner. A *wrapper* is simply a procedure, customized to a particular information resource, for extracting the content from such pages.

We then described *inductive learning*, our approach to automatically learning wrappers (Chapter 3). Induction is the process of reasoning from a set of *examples* to some *hypothesis* that (in some application-specific sense) explains or generalizes over the examples. For example, if told that '*Thatcher lied*', '*Mao lied*', and '*Einstein didn't lie*', an inductive learner might hypothesize that '*Politicians lie*'. Each example consists of an *instance* and its *label* according to the *target* hypothesis. Induction is thus a matter of reconstructing the target from a sample of its behavior on a number of example instances. We defined the Induce generic learning algorithm. To learn a particular hypothesis is some particular class $\mathcal{H}$, Induce is provided with an *oracle*, which provides a stream of examples, as well as a *generalization function* $\mathsf{Generalize}_{\mathcal{H}}$ that is specialized to the class $\mathcal{H}$.

In our application, examples correspond to the query responses, and hypotheses correspond to wrappers. The example pages are labeled with the content that is to be extracted from them. Describing how to learn wrappers thus mainly involves defining a class $\mathcal{W}$ of wrappers and the implementing the function $\mathsf{Generalize}_{\mathcal{W}}$.

Contribution 2: Reasonably expressive yet efficiently learnable wrapper classes. In Chapter 4 we describe one such class, HLRT. As discussed in Section 4.2, *h̲ead-l̲eft-r̲ight-t̲ail* wrappers formalize a simple but common "programming idiom". HLRT wrappers operate by scanning the page for specific constraint strings that separate the body of the page from its head and tail (which might contain extraneous but confusing text), as well as strings that delimit each text fragment to be extracted. Though simple, our empirical results indicate that HLRT wrappers can handle 57% of a recently surveyed list of actual Internet resources (Section 7.2).

In Section 4.3, we proceeded by defining the $\mathsf{Generalize}_{\mathrm{HLRT}}$ function. To do so, we specified a set of constraints (denoted $\mathcal{C}_{\mathrm{HLRT}}$) that must hold if an HLRT wrapper is to operate correctly on a given set of examples. Thus $\mathsf{Generalize}_{\mathrm{HLRT}}$ is a special-purpose constraint satisfaction engine, customized to solving the constraint $\mathcal{C}_{\mathrm{HLRT}}$.

Our naïve generate-and-test implementation of $\mathsf{Generalize}_{\mathrm{HLRT}}$ is relatively simple. Unfortunately, this algorithm runs very slowly; our worst-case complexity analysis shows that it runs in time that is *exponential* in the number of attributes (Theorem 4.4). But a careful examination of $\mathcal{C}_{\mathrm{HLRT}}$ reveals that it can be decomposed into several distinct subproblems. In Section 4.4, we use this analysis to develop the $\mathsf{Generalize}^{*}_{\mathrm{HLRT}}$ algorithm, which runs in time that is *linear* in the number of attributes (Theorem 4.6).

Our experiments in Section 7.3 demonstrate that $\mathsf{Generalize}^{*}_{\mathrm{HLRT}}$ is quite fast in practice. Indeed, this performance is somewhat at odds with Theorem 4.6's complexity bound: while the bound is linear in the number of attributes, it is actually a degree-six polynomial in the size of the examples. Our heuristic-case analysis (Theorem 4.7 in Section 4.5) explains analytically why $\mathsf{Generalize}^*_{\mathrm{HLRT}}$ is so fast in practice. In Section 7.5, we empirically validated the assumptions underlying this analysis.

In any inductive learning setting, running time is only one important resource. The number of examples required for effective learning is often even more significant. For instance, in our wrapper-construction application, each example consumes substantial network bandwidth. The *probably approximately correct* (PAC) model of inductive learning an approach to obtaining theoretical bounds on the number of examples needed to satisfy a user-specified level of performance. In Section 3.2.3 we described the PAC model in general, and in Section 4.6 we developed a PAC model for the HLRT wrapper class. The main result is that—under assumptions that are empirically validated (Section 7.6)—the PAC model's prediction of the number of training examples required for the HLRT class is *independent* of the number of attributes, and *polynomial* in all relevant parameters (Theorem 4.9).

Finally, HLRT is just one of many possible wrapper classes. In Chapter 5, we define five more classes, LR, OCLR, HOCLRT, N-LR and N-HLRT. Adopting the framework developed for the HLRT class, we describe the $\mathsf{Generalize}_{\mathcal{W}}$ function for each of these five wrapper classes. We then compare all six classes on two grounds. First (Theorems 5.1 and 5.10), we analyze the *relative expressiveness* of the six wrapper classes, which characterize the extent to which wrappers in one class can mimic the behavior of another. Second (Theorems 4.7, 5.4, 5.7, 5.9, 5.12 and 5.14) we analyzed the heuristic-case running time of each $\mathsf{Generalize}_{\mathcal{W}}$ function.

Contribution 3. Noise-tolerant techniques for automatically labeling examples. Our inductive approach to constructing wrappers requires a supply of *labeled* example pages. In Chapter 6, we describe techniques to automate this labeling process. The basic idea is to take as input a library of *recognizers*. A recognizer is a procedure that examines a page for instances of a particular attribute. For example, when learning a wrapper for the country/code resource, the system takes as input a recognizer for the countries, and another recognizer for the codes. The results of these recognizers are then integrated to generate a label for the entire page.

Clearly, this *corroboration* process is trivial if each of the recognizers is *perfect*. Assuming perfect recognizers is unrealistic, though corroboration is very difficult unless the library contains at least one perfect recognizer, which can be used to anchor the evidence provided by the others. Corroboration is also difficult if any of the recognizers are *unreliable* (*i.e.*, might report both spurious false positive instances as well as miss some true positives). We developed the Corrob algorithm (Section 6.4), which can handle recognizer libraries that contain at least one perfect recognizer, but no unreliable recognizers. (Note that though the recognizers can not be unreliable, they can be *unsound* (exhibit false positives but no false negatives) or *incomplete* (false negatives but no false positives).)

The idea, then, is to use Corrob to label the example pages. But in general, recognizers give ambiguous or incomplete information about a page's label. Thus we face an interesting problem of learning from *noisily-labeled pages*. Fortunately, as we demonstrate for the HLRT class in Section 6.5, it is relatively straightforward to modify the Generalize$_{\mathcal{W}}$ functions and PAC model to handle this noise. Our approach to learning from noisy data is somewhat unusual: our wrapper induction application is "self-correcting", in that it can usually determine which labels are correct.

Unfortunately, Corrob is very slow unless all the recognizers are perfect. We thus developed Corrob* (Section 6.6), which heuristically solves a simplified version of the corroboration problem.

9.2 Future work

While this summary describes the progress we have made, in many ways our techniques for automatic wrapper construction just scratch the surface. We now suggest promising directions for future research.

9.2.1 Short-to-medium term ideas

The point of our effort is to build tools to make it easier to write wrappers for actual Internet information resources. While our experiments and analysis demonstrate that our techniques are effective for such resources, there are numerous ways to make our system more immediately useful and practical.

First, there are many subtle user-interface issues that must be handled when developing a useful wrapper-construction tool. Our WIEN application (Section 7.8) application could be improved in several ways. First, we have many ideas for how to change the user interface to simplify the process of wrapper construction. For example, we are experimenting with *aggressive learning*, in which WIEN starts learning as soon as parts of the page are labeled. Of course the results of such learning might be wrong, and thus we must provide a mechanism for easily correcting WIEN's mistakes. Second, our six wrapper classes are rather general purpose, but they are unknown outside our research group; it would be interesting to select a standard wrapper language (*e.g.*, the wrapper languages discussed in [Ashish & Knoblock 97a, Roth & Schwartz 97, Chidlovskii et al. 97]) and extend WIEN so that it builds wrappers that adhere to an emerging standard.

Second, as discussed in Section 6.7, we have not expended much effort developing recognizers. To make our techniques more useful, it would be helpful to develop a library of common recognizers. While many recognizers would no doubt be simply idiosyncratic hacks, it would also be worthwhile trying to develop a more general framework.

A third important direction, related to the second, is the elaboration of our work on corroboration. We developed the Corrob* algorithm, which works very well in the domains we have investigated. But recall that Corrob* makes several simplifications and assumptions (Section 6.6). We are a long way from efficient corroboration algorithms that work well with unrestricted recognizer libraries.

A fourth extension would be to investigate ways to make the generalization functions ($\mathsf{Generalize}_{\mathcal{W}}$ for each class $\mathcal{W}$) run faster. One idea would be to investigate the *search control* used by these functions. Figure 4.5 on page 54 shows the space searched by the $\mathsf{Generalize}_{\mathrm{HLRT}}$ function. We did not specify exactly how this space is traversed, beyond stating that it is searched in a depth-first manner. Recall that $\mathsf{Generalize}_{\mathrm{HLRT}}$ tries each candidate for each HLRT component; *e.g.*, line 4.6(b) tries all candidates for r_k. Different orderings of these candidates correspond to different ways to traverse the space of HLRT wrappers. As our experiments in Section 7.3 show, the default search order is reasonably fast. However, more advanced search control would probably speed up the search.

A final direction for short-to-medium term future work involves the wrapper classes themselves. We have identified six wrapper classes; they are reasonably useful for actual Internet resources and yet can be learned reasonably quickly. But obviously there are many other classes that can be considered. We have two specific suggestions.

- Our coverage survey (Section 7.2) suggests that the N-LR and N-HLRT wrapper classes are less useful than anticipated; N-LR, for example, can handle just 13% of the surveyed sites. The problem is that these classes offer tremendous flexibility; they allow for *arbitrarily* nested pages. But this flexibility requires that the pages be highly structured as well: there must exist a set of delimiters that reliably extract this structure, and the constraints on these delimiters are much more stringent than for the tabular wrappers. Unfortunately, the data suggest that relatively few sites have such structure. Thus one fruitful avenue for future work would be a more careful investigation of the issues related to building wrappers for hierarchically nested resources.

- Our results in Section 7.2 indicate that about 30% of the surveyed information resources can not be handled by any of the six wrapper classes we discuss. It would be helpful to develop wrapper classes for these resources. Inspection of these sites reveals that many contain *missing attributes*—*e.g.*, in the country/code resource, perhaps the country code is occasionally missing. None of the six wrapper classes we defined can handle missing attributes[1] Our preliminary investigation suggests that about 20% (*i.e.*, about two-thirds of the unwrappable sites) could benefit from wrappers that can handle missing values.

  Our delimiter-based wrapper framework can be extended to handle missing attributes. Consider M-LR, a simple extension to the LR wrapper class. Like LR, M-LR uses two delimiters, ℓ_k and r_k, to indicate the beginning and end of each attribute. M-LR wrappers also use a third delimiter per attribute, x_k, which indicates that the attribute is missing. For example, if countries are formatted as `<B>Congo</B>` or as `<BLINK>Missing</BLINK>` if missing, then an M-LR wrapper could ℓ_k = `<B>`, r_k = `</B>` and x_k = `<BLINK>`. While we have not investigated the coverage of the M-LR class (a la Section 7.2), it is nonetheless usfseful to ask how quickly M-LR wrappers can be learned.

  Note that for each k, delimiters ℓ_k and x_k interact (just as h, t and ℓ_1 interact for the HLRT class). Therefore learning M-LR is slower than learning LR, since the $\mathsf{Generalize}_{\text{M-LR}}$ function must enumerate the combinations of candidates for ℓ_k and x_k. (As with LR, the r_k delimiters can be learned in isolation). To

[1] Actually, the N-LR and N-HLRT wrapper class can handle a special case of missing attributes: if K attributes are to be extracted, then attributes K, $K-1$, $K-2$, …—*i.e.*, the "bottom" several layers of nodes in a tree such as Figure 5.2 on page 92—can be missing.

summarize, we conjecture that learning M-LR is harder than learning LR, though easier than OCLR, HLRT, HOCLRT, N-LR and N-HLRT.

9.2.2 Medium-to-long term ideas

There are several directions in which to push our work that concern the "bigger picture". While our work on wrapper construction is an important enabling technology for a variety of information integration applications, there are still many problems left unsolved.

Consider first the task of information extraction from semi-structured resources. We emphasized one simple version of this task: information is to be extracted from a *single* document containing tuples from *one* relation. As a concrete example, the information from a query to the country/code resource is contained in a single page, and there is just one table of data on this page (namely, the country/code table). But industrial-strength wrapper applications require more sophisticated functionality [Doorenbos 97], such as

- extracting and merging data from multiple documents—*e.g.*, clicking on a table of hyper-linked product names in order to get their prices; and

- extracting information from a single page that contains more than one collection of data—*e.g.*, a page might contain one table listing products and prices, and another table listing manufacturers and their telephone numbers.

It is straightforward to compose our wrappers so that such issues can be handled: a *meta-wrapper* can invoke a wrapper on the pages obtained by clicking on hyper-links, or invokes several wrappers on the same page. However, *automatically* constructing such wrappers is a challenging direction for future work.

A second important long-term research direction involves closing information integration loop. We have examined *information extraction* in isolation, but our techniques must be integrated with work on *resource discovery* [Bowman et al. 94, Zaiane & Jiawei 95], learning to *query* information resources [Cohen & Singer 96, Doorenbos et al. 97], and learning *semantic models* of information resources [Perkowitz & Etzioni 95, Tejada et al. 96].

Third, our work has focused on resources whose content is formatted by HTML tags. Let us emphasize that our techniques do *not* depend on HTML or any other particular formatting convention. Nevertheless, we have not yet demonstrated that our techniques work for other kinds of semi-structured information resources. For example, in Section 8.2.2, we mentioned that it would be interesting to apply our techniques to legacy information systems.

Another possibility is to apply our techniques to various formats used to encode tabular information, such as CSV or DIF (two spreadsheet standards), or various document standards such as "Rich Text Format" (RTF) or LaTeX. For instance, the following code samples:

standard	example

HTML	<TABLE>
	  <TR><TD>A</TD><TD>B</TD><TD>C</TD></TR>
	  <TR><TD>D</TD><TD>E</TD><TD>F</TD></TR>
	</TABLE>

CSV	A,B,C D,E,F
DIF	DATA BOT 1,0 A 1,0 B 1,0 C BOT 1,0 D 1,0 E 1,0 F EOD
RTF	\trowd A \cell B \cell C \cell \row \trowd D \cell E \cell F \cell \row

LaTeX	\begin{tabular}{ccc}
	  A & B & C \\
	  D & E & F \\
	\end{tabular}

all encode the following table:

A	B	C
D	E	F

Clearly, most if not all of these formats can be handled by the sort of delimiter-based wrappers discussed in this thesis.

Finally, as a fourth medium-to-long term direction for future research, we propose integrating our techniques with traditional natural language processing approaches to information extraction (IE). Effective IE systems should leverage whatever structure is available in the text, whether it be linguistic or extra-linguistic. Ideally, inductive learning techniques can be used to discover such regularities automatically.

This thesis demonstrates that such an approach is feasible for the kinds of semi-structured documents found on the Internet. At the other end of the spectrum, AUTOSLOG [Riloff 93] and CRYSTAL/WEBFOOT [Soderland et al. 95, Soderland 97b, Soderland 97c] have demonstrated the feasibility of this approach for natural language text. More recently, Soderland has developed WHISK [Soderland 97a], which learns information extraction rules in highly structured but non-grammatical domains such as apartment listings. WHISK's rules are essentially regular expressions, specifying specific delimiters, quite similar to our delimiter-based wrappers. But truly general purpose IE systems are still a long way off.

9.2.3 Theoretical directions

Finally, let us mention two directions in which our theoretical analysis can be extended.

First, as discussed in Section 7.4, our PAC model is too loose by one to two orders of magnitude. Thus we can not realistically use this PAC model to automatically terminate the learning algorithm. Tightening this model would be an interesting direction for future work. Note that the PAC model makes worst-case assumptions about the learning task. Specifically, it assumes that the distribution $\mathcal{D}$

over examples is arbitrary. A standard technique for tightening a PAC model is to assume that $\mathcal{D}$ has certain properties [Benedek & Itai 88, Bartlett & Williamson 91]. [Schuurmans & Greiner 95] suggests another strategy: by replacing the "batch" model on inductive learning with a "sequential" model in which the PAC-theoretic analysis is repeated as each example is observed, many fewer examples are predicted. It would be interesting to apply each of these approaches to our wrapper construction task.

Second, although we are interested in completely automating the wrapper construction process, realism demands that we focus on semi-automatic systems for some time to come. But our techniques do not take into account the cost of asking a person for assistance. For example, if a person plays the role of a recognizer for country names, then our system requires that the person locate all the countries in the documents, even though it is possible that pointing out just one or two countries could be sufficient for the system to guess the rest. Our investigation of aggressive learning (Section 9.2.1) is a step in the right direction. But it would be interesting to develop a model of learning that explicitly reasons about the utility of asking the oracle various questions.

9.3 Conclusions

The Internet presents a dizzying array of information resources, and more are coming on-line every day. Many of these sites—those whose goal is purely entertainment, or the growing collection of on-line magazines and newspapers—are perhaps suited only for direct manual browsing. But many other sites—airline schedules, retail product catalogs, weather forecasts, stock market quotations—provide structured data. As the number and variety of such resources has grown, many have argued for the development of systems that automatically interact with such resources in order to extract this structured content on a user's behalf.

There are many technical challenges to building such systems, but one in particular is immediately apparent as soon as one starts to design such a system. There is a huge amount of information *available* on the Internet, but machines today *understand* very little of it. While standards such as XML or Z39.50 promise to ameliorate this situation, standards require cooperation or enforcement, both of which are in short supply on the Internet today.

In this thesis, we have investigated techniques—wrapper learning algorithms, and corroboration algorithms for generating labeled examples—that promise to significantly increase the amount of on-line information to which software agents have access. Although our techniques make relatively strong assumptions, we have attempted to empirically validate these assumptions against actual Internet resources.

As described earlier in this chapter, there are many open issues. But we think that this thesis demonstrates that automatic wrapper construction can facilitate the construction of a variety of software agents and information integration tools.

Appendix A

AN EXAMPLE RESOURCE AND ITS WRAPPERS

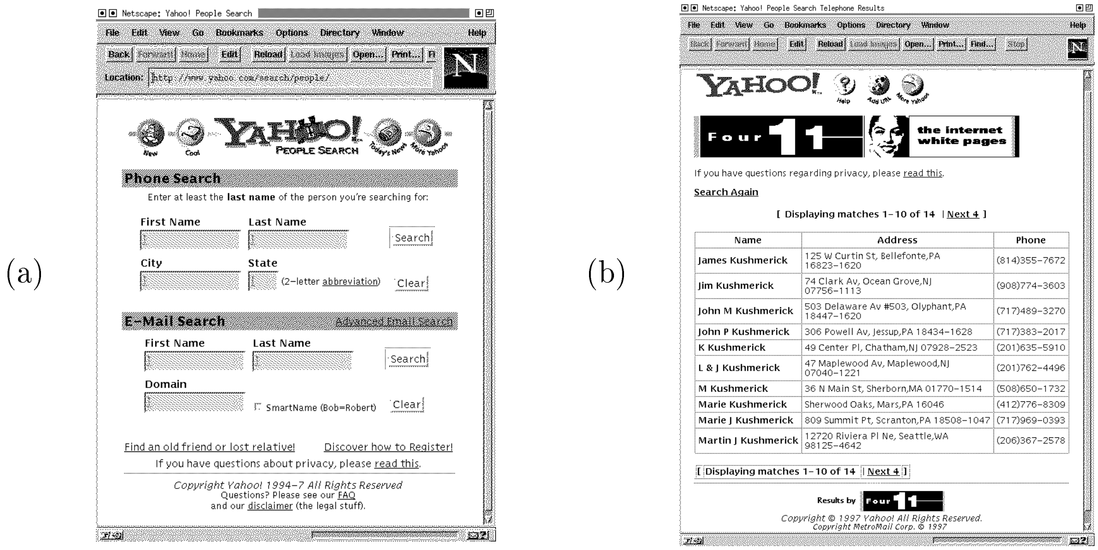
In this Appendix, we provide a concrete instance of each of the six wrapper classes discussed in this thesis: LR, HLRT, OCLR, HOCLRT, N-LR and N-HLRT. We provide examples of the HTML source for site 4 in the "www.search.com" survey described in Section 7.2: "Yahoo People Search: Telephone/Address", available at `http://www.yahoo.com/search/people/`. Site 4 can be wrapped by all six wrapper classes, and so we present a wrapper in each class that is consistent with pages drawn from this resource.

Resource 4. To begin, Figure A.1(a) shows the query interface for resource 4, while Figure A.1(b) provides an example of the resource's query responses. The site is treated as yielding tuples containing $K = 4$ attributes (name, address, area code, and phone number).

Example 1. We now list the HTML source for two of the example query responses. The HTML is shown exactly as supplied from the resource, except that white-space has been altered in unimportant ways to simplify the presentation. The information content of each page is indicated with $\boxed{\text{boxes}}$ around the text to be extracted.

The first example is the HTML source of the response to the query "Last Name = kushmerick", depicted in Figure A.1(b).

```
<head>
<title>
Yahoo!  People Search Telephone Results
</title>
</head>
<body>
```


(a)
Netscape: Yahoo! People Search
File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Edit Reload Load Images Open... Print... Fi
Location: http://www.yahoo.com/search/people/
New
Cool
YAHOO! PEOPLE SEARCH
Today's News
More Yahoos
Search
Clear
Search
Clear
(b)
Netscape: Yahoo! People Search Telephone Results
File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Edit Reload Load Images Open... Print... Find... Stop
YAHOO!
Help
Add URL
More Yahoos
Four 11
the internet white pages
Results by Four 11

Phone Search

Enter at least the **last name** of the person you're searching for:

First Name

Last Name

City

State (2-letter abbreviation)

E-Mail Search Advanced Email Search

First Name

Last Name

Domain

□ SmartName (Bob=Robert)

Find an old friend or lost relative! Discover how to Register!

If you have questions about privacy, please read this.

Copyright Yahoo! 1994–7 All Rights Reserved
Questions? Please see our FAQ
and our disclaimer (the legal stuff).

If you have questions regarding privacy, please read this.

Search Again

[Displaying matches 1–10 of 14 | Next 4]

Name	Address	Phone
James Kushmerick	125 W Curtin St, Bellefonte,PA 16823–1620	(814)355–7672
Jim Kushmerick	74 Clark Av, Ocean Grove,NJ 07756–1113	(908)774–3603
John M Kushmerick	503 Delaware Av #503, Olyphant,PA 18447–1620	(717)489–3270
John P Kushmerick	306 Powell Av, Jessup,PA 18434–1628	(717)383–2017
K Kushmerick	49 Center Pl, Chatham,NJ 07928–2523	(201)635–5910
L & J Kushmerick	47 Maplewood Av, Maplewood,NJ 07040–1221	(201)762–4496
M Kushmerick	36 N Main St, Sherborn,MA 01770–1514	(508)650–1732
Marie Kushmerick	Sherwood Oaks, Mars,PA 16046	(412)776–8309
Marie J Kushmerick	809 Summit Pt, Scranton,PA 18508–1047	(717)969–0393
Martin J Kushmerick	12720 Riviera Pl Ne, Seattle,WA 98125–4642	(206)367–2578

[Displaying matches 1–10 of 14 | Next 4]

Copyright © 1997 Yahoo! All Rights Reserved.
Copyright MetroMail Corp. © 1997

Figure A.1: *Site 4 in the survey (Figure 7.1):* (a) *the query interface; and* (b) *an example query response.*

```
<!--Yahoo Subcategory Banner-->
<map name="menu">
<area shape=rect coords="0,0,210,56" href="http://www.yahoo.com/">
<area shape=rect coords="210,0,265,56"
href="http://www.yahoo.com/docs/info/help.html">
<area shape=rect coords="265,0,317,56" href="http://add.yahoo.com/bin/add?">
<area shape=rect coords="319,0,446,56"
href="http://www.yahoo.com/docs/family/more.html">
</map>
<img src="http://www.yahoo.com/images/cat3.gif" border=0 usemap="#menu" ismap>
<p><!-- Default Fail --><IMG SRC=/g/rlogo.gif ALT="Four11 Corp" HEIGHT=66
WIDTH=515>
<P>
If you have questions regarding privacy, please
<a href="http://www.yahoo.com/docs/info/people_privacy.html">read this</a>.<p>
<a href="http://www.yahoo.com/search/people"><b>Search Again</b></a><p>
<center>
<table><tr>
<td><b>[</b></td>
<td>
<b>Displaying matches 1-10 of 14</b>
</td>
<td>| <strong><a href=/cgi-bin/Four11?YahooPhoneResults&Offset=104464&Id=2&
NameCount=10&LastName=kushmerick&FirstName=&City=&State=> Next 4</a></strong></td>
<td><b>]</b></td>
</tr></table>
<p>
```

```
<table border=1 cellpadding=4 cellspacing=0
WIDTH=100<tr><th>Name<th>Address<th>Phone</tr>
<tr>
<td><strong>James Kushmerick</strong></td>
<td>125 W Curtin St, Bellefonte,PA 16823-1620</td>
<td>(814)355-7672</td>
</tr>
<tr>
<td><strong>Jim Kushmerick</strong></td>
<td>74 Clark Av, Ocean Grove,NJ 07756-1113</td>
<td>(908)774-3603</td>
</tr>
<tr>
<td><strong>John M Kushmerick</strong></td>
<td>503 Delaware Av #503, Olyphant,PA 18447-1620</td>
<td>(717)489-3270</td>
</tr>
<tr>
<td><strong>John P Kushmerick</strong></td>
<td>306 Powell Av, Jessup,PA 18434-1628</td>
<td>(717)383-2017</td>
</tr>
<tr>
<td><strong>K Kushmerick</strong></td>
<td>49 Center Pl, Chatham,NJ 07928-2523</td>
<td>(201)635-5910</td>
</tr>
<tr>
<td><strong>L & J Kushmerick</strong></td>
<td>47 Maplewood Av, Maplewood,NJ 07040-1221</td>
<td>(201)762-4496</td>
</tr>
<tr>
<td><strong>M Kushmerick</strong></td>
<td>36 N Main St, Sherborn,MA 01770-1514</td>
<td>(508)650-1732</td>
</tr>
<tr>
<td><strong>Marie Kushmerick</strong></td>
<td>Sherwood Oaks, Mars,PA 16046</td>
<td>(412)776-8309</td>
</tr>
<tr>
<td><strong>Marie J Kushmerick</strong></td>
<td>809 Summit Pt, Scranton,PA 18508-1047</td>
<td>(717)969-0393</td>
</tr>
<tr>
<td><strong>Martin J Kushmerick</strong></td>
<td>12720 Riviera Pl Ne, Seattle,WA 98125-4642</td>
<td>(206)367-2578</td>
</tr>
```

```
</table>
<br>
</center>
<table border=1><tr>
<td><b>[</b></td>
<td>
<b>Displaying matches 1-10 of 14</b>
</td>
<td>| <strong><a href=/cgi-bin/Four11?YahooPhoneResults&Offset=104464&Id=2&
NameCount=10&LastName=kushmerick&FirstName=&City=&State=> Next 4</a></strong></td>
<td><b>]</b></td>
</tr></table>
<HR>
<center>
<img src="http://images.Four11.com/g/Yahoo/resultsbyfour11.gif"
alt="results by Four11" width=207 height=39><BR>
<i>Copyright &copy; 1997 Yahoo!  All Rights Reserved.<br>
<FONT SIZE=-1> Copyright MetroMail Corp.  &copy; 1997 </FONT>
</center>
</body>
```

Example 2. The second example is the HTML source of the response to the query "First Name = weld".

```
<head>
<title>
Yahoo!  People Search Telephone Results
</title>
</head>
<body>
<!--Yahoo Subcategory Banner-->
<map name="menu">
<area shape=rect coords="0,0,210,56" href="http://www.yahoo.com/">
<area shape=rect coords="210,0,265,56"
href="http://www.yahoo.com/docs/info/help.html">
<area shape=rect coords="265,0,317,56" href="http://add.yahoo.com/bin/add?">
<area shape=rect coords="319,0,446,56"
href="http://www.yahoo.com/docs/family/more.html">
</map>
<img src="http://www.yahoo.com/images/cat3.gif" border=0 usemap="#menu" ismap>
<p><!-- Default Fail --><IMG SRC=/g/rlogo.gif ALT="Four11 Corp" HEIGHT=66
WIDTH=515>
<P>
If you have questions regarding privacy, please <a
href="http://www.yahoo.com/docs/info/people_privacy.html">read this</a>.<p>
<a href="http://www.yahoo.com/search/people"><b>Search Again</b></a><p>
There are <b>over 200</b> names that meet your search criteria.<br>
Try narrowing your search by specifying more parameters.
Here are the <b>first 200</b> names.<P>
```

```
<center>
<table><tr>
<td><b>[</b></td>
<td>
<b>Displaying matches 1-10 of 200</b>
</td>
<td>| <strong><a href=/cgi-bin/Four11?YahooPhoneResults&Offset=896&Id=1&
NameCount=10&LastName=&FirstName=weld&City=&State=> Next 10</a></strong></td>
<td><b>]</b></td>
</tr></table>
<p>
<table border=1 cellpadding=4 cellspacing=0
WIDTH=100<tr><th>Name<th>Address<th>Phone</tr>
<tr>
<td><strong>Weld Birmingham</strong></td>
<td>28 22nd Av Nw, Birmingham,AL 35215-3412</td>
<td>(205)854-1688</td>
</tr>
<tr>
<td><strong>Weld Butler</strong></td>
<td>Cider Hl Rd, York,ME 03909</td>
<td>(207)363-6104</td>
</tr>
<tr>
<td><strong>Weld & Mary Butler</strong></td>
<td>11 Kings, Eliot,ME 03903</td>
<td>(207)439-5137</td>
</tr>
<tr>
<td><strong>Weld S & Jessie Carter</strong></td>
<td>7405 Cresthill Ct, Fox Lake,IL 60020-1008</td>
<td>(847)587-9890</td>
</tr>
<tr>
<td><strong>Weld S & Ruth Carter</strong></td>
<td>106 Lorimer Rd, Belmont,MA 02178-1004</td>
<td>(617)484-2027</td>
</tr>
<tr>
<td><strong>Robert & Weld Conley</strong></td>
<td>312 S Beckley Sta Rd, Louisville,KY 40245-4002</td>
<td>(502)254-2002</td>
</tr>
<tr>
<td><strong>Weld Conley</strong></td>
<td>804 Corona Ct, Louisville,KY 40222</td>
<td>(502)425-3787</td>
</tr>
<tr>
<td><strong>Weld Coxe</strong></td>
<td>44 Concord Av #502, Cambridge,MA 02138-2350</td>
```

```
<td>(617) 492-8622</td>
</tr>
<tr>
<td><strong>Weld H Fickel</strong></td>
<td>Kernville #963, Kernville,CA 93238-0963</td>
<td>(619) 376-2366</td>
</tr>
<tr>
<td><strong>Weld Field</strong></td>
<td> Columbia,SC 29210</td>
<td>(803) 750-0001</td>
</tr>
</table>
<br>
</center>
<table border=1><tr>
<td><b>[</b></td>
<td>
<b>Displaying matches 1-10 of 200</b>
</td>
<td>| <strong><a href=/cgi-bin/Four11?YahooPhoneResults&Offset=896&Id=1&
NameCount=10&LastName=&FirstName=weld&City=&State=> Next 10</a></strong></td>
<td><b>]</b></td>
</tr></table>
<HR>
<center>
<img src="http://images.Four11.com/g/Yahoo/resultsbyfour11.gif"
alt="results by Four11" width=207 height=39><BR>
<i>Copyright &copy; 1997 Yahoo!  All Rights Reserved.<br>
<FONT SIZE=-1> Copyright MetroMail Corp.  &copy; 1997 </FONT>
</center>
</body>
```

The wrappers. The following wrappers are consistent with these examples (as well as eight more collected from this resource):

class	wrapper
LR and N-LR	$\langle \ell_1, r_1, \ell_2, r_2, \ell_3, r_2, \ell_4, r_4 \rangle$
HLRT and N-HLRT	$\langle h, t, \ell_1, r_1, \ell_2, r_2, \ell_3, r_2, \ell_4, r_4 \rangle$
OCLR	$\langle o, c, \ell_1, r_1, \ell_2, r_2, \ell_3, r_2, \ell_4, r_4 \rangle$
HOCLRT	$\langle h, t, o, c, \ell_1, r_1, \ell_2, r_2, \ell_3, r_2, \ell_4, r_4 \rangle$

where

$$h = \texttt{<body>}$$

$$\begin{aligned}
t &= \texttt{</body>} \\
o &= \texttt{<td><strong>} \\
c &= \phi \\
\ell_1 &= \texttt{<td><strong>} \\
\ell_2 &= \texttt{<td>}\Downarrow\texttt{<td>} \\
\ell_3 &= \texttt{(} \\
\ell_4 &= \texttt{)} \\
r_1 &= \texttt{</strong></td>} \\
r_2 &= \texttt{</td>} \\
r_3 &= \texttt{)} \\
r_4 &= \texttt{</td>}
\end{aligned}$$

(Recall from Appendix C that ϕ denotes the empty string and $\Downarrow$ is the carriage return character.)

Note that these particular wrappers are one of many possible for each class. For example, exhaustive enumeration reveals that there are more than twenty-five million consistent LR wrappers.

Appendix B

PROOFS

B.1 Proof of Theorem 4.1

Proof of Theorem 4.1: We must prove that, for any wrapper $w = \langle h, t, \ell_1, r_1, \ldots, \ell_K, r_k \rangle$ and example $\langle P, L \rangle$, $\mathcal{C}_{\mathrm{HLRT}}(w, \langle P, L \rangle) \iff \mathsf{ExecHLRT}(w, P) = L$. (As usual, let $M = |L|$ be the number of tuples in the example, and let K be the number of attributes per tuple.)

Part I ($\Rightarrow$). We demonstrate that if $\mathcal{C}_{\mathrm{HLRT}}(w, \langle P, L \rangle)$, then:

1. $b_{1,1}$ (the first tuple's first attribute's beginning index) is calculated correctly (*i.e.*, the computed $b_{1,1}$ equals the corresponding value in P's label L).
2. For each $1 \leq k \leq K$ and $1 \leq m \leq M$, if index $b_{m,k}$ is calculated correctly, then the end index $e_{m,k}$ is calculated correctly.
3. For each $1 \leq k < K$ and $1 \leq m \leq M$, if $e_{m,k}$ is calculated correctly, then $b_{m,k+1}$ is calculated correctly.
4. For each $1 \leq m < M$, if $e_{m,K}$ is calculated correctly, then $b_{m+1,1}$ is calculated correctly.
5. If $e_{M,K}$ is calculated correctly, then ExecHLRT will immediately halt.

Note that establishing each of these claims suffices to establish Part I.

1. ExecHLRT calculates $b_{1,1}$ by setting i to the position of h in P (line 4.1(a)), and then setting $b_{1,1}$ to the first character following the next occurrence of ℓ_1 (lines 4.1(c–d)). Since $\mathcal{C}_{\mathrm{HLRT}}(w, \langle P, L \rangle)$ holds, we know that $\mathbf{C3}(h, t, \ell_1, \langle P, L \rangle)$ holds. In particular, we know that **C3(i–ii)** both hold. **C3(i)** guarantees that this procedure works properly: since ℓ_1 is a proper suffix of the part of page

P's head following h, searching for h and then ℓ_1 will in fact find the end of the head. Moreover, **C3(ii)** guarantees that the test at line 4.1(b) succeeds, so that line 4.1(c) is actually reached.

2. ExecHLRT calculates $e_{m,k}$ by starting with index i equal to $b_{m,k}$ (line 4.1(d)) and then searching for the next occurrence of r_k. Since $\mathcal{C}_{\text{HLRT}}(w, \langle P, L\rangle)$ holds, we know that $\mathbf{C1}(r_k, \langle P, L\rangle)$ holds. Therefore, r_k will *not* be found in the attribute value itself (**C1(ii)**), but *will* be found immediately following the attribute value (**C1(i)**). Therefore, searching for string r_k at position $b_{m,k}$ does in fact find index $e_{m,k}$.

3. ExecHLRT calculates $b_{m,k+1}$ ($k < K$) by starting with index i equal to the position of r_k (line 4.1(e)), and then setting $b_{m,k+1}$ equal to the next occurrence of ℓ_{k+1} (line 4.1(c–d)). Since $\mathcal{C}_{\text{HLRT}}(w, \langle P, L\rangle)$ holds, we know that $\mathbf{C2}(\ell_{k+1}, \langle P, L\rangle)$ holds. Therefore, ℓ_{k+1} is a proper suffix of the page fragment between i and the beginning of the next tuple (which has index $b_{m,k+1}$). Thus ExecHLRT will indeed find $b_{m,k+1}$ properly.

4. ExecHLRT calculates $b_{m+1,1}$ ($m < M$) by starting with the index i pointing at the end of the previous tuple, which has index $e_{m,K}$ (line 4.1(e) during the $K^{\underline{\text{th}}}$ iteration of the inner 'for each' loop). Then, $b_{m+1,1}$ is calculated by scanning forward for the next occurrence of ℓ_1. Since $\mathcal{C}_{\text{HLRT}}(w, \langle P, L\rangle)$ holds, we know that $\mathbf{C3}(h, t, \ell_1, \langle P, L\rangle)$ holds. In particular, we know that **C3(iv)** (which guarantees that this procedure works properly) and **C3(v)** (which guarantees that the test at line 4.1(b) succeeds, so that line 4.1(c) is actually reached) both hold.

5. After calculating $e_{M,K}$ correctly, the index i point to the first character of page P's tail, and ExecHLRT invokes the termination condition test at line

4.1(b). Since $\mathcal{C}_{\mathrm{HLRT}}(w, \langle P, L\rangle)$ holds, we know that $\mathbf{C3}(h, t, \ell_1, \langle P, L\rangle)$ holds. In particular, we know that **C3(iii)** holds, which ensures that this termination condition fails, so ExecHLRT halts.

To summarize, we have shown that label L's $b_{m,k}$ and $e_{m,k}$ values are all computed correctly, with no content being incorrectly extracted, no content being skipped, and no "extra" content being extracted.

Part II ($\Leftarrow$). We demonstrate a contradiction. Suppose there exists an example $\langle P, L\rangle$ such that $\mathsf{ExecHLRT}(w, P) = L$, yet $\neg\mathcal{C}_{\mathrm{HLRT}}(w, \langle P, L\rangle)$. Then one of the three constraints **C1**–**C3** must be incorrect—*i.e.*, it must be the case that $\mathsf{ExecHLRT}(w, P) = L$ even though one of **C1**–**C3** doesn't hold.

C1: For each k, if $\neg\mathbf{C1}(r_k, \langle P, L\rangle)$, then ExecHLRT's line 4.1(f) would have calculated $e_{m,k}$ incorrectly for at least one value of m. But we assumed that $\mathsf{ExecHLRT}(w, P) = L$, and thus the $e_{m,k}$ must be correct.

C2: For each k, if $\neg\mathbf{C2}(\ell_k, \langle P, L\rangle)$, then $S_{m,k-1}/\ell_k \neq \ell_k$ for some particular tuple m. But then ExecHLRT's line 4.1(d) would have calculated $b_{m,k}$ incorrectly. But we assumed that $\mathsf{ExecHLRT}(w, P) = L$, and thus $b_{m,k}$ must be correct.

C3: Constraint **C3** has five parts:

C3(i) If this predicate does not hold, then the execution of line 4.1(a) would set $i \leftarrow \Diamond$. The behavior of ExecHLRT under these circumstances is unspecified, but we can assume that $\mathsf{ExecHLRT}(w, P) \neq L$.[1]

C3(ii) If this predicate fails, then line 4.1(c) would fail to calculate $b_{1,1}$ correctly, or the outer loop (line 4.1(b)) would iterate too few times.

C3(iii) If this predicate fails, then the outer loop (line 4.1(b)) would iterate too many times.

[1] Throughout this thesis, we ignore the issue of applying a wrapper to an inappropriate page. These circumstances are of little theoretical significance; for example, we could easily have specified formally what happens if $i \leftarrow \Diamond$, but that would have simply cluttered the algorithms while providing little benefit. Moreover, since these modifications are so simple, we do not need to be concerned about them from a practical perspective either.

C3(iv) If this predicate fails for the $m^{\underline{\text{th}}}$ tuple, then line 4.1(c) would fail to calculate $b_{m,1}$ correctly.

C3(v) If this predicate fails for the $m^{\underline{\text{th}}}$ tuple, then the outer loop (line 4.1(b)) would stop after m iterations instead of M.

□ (Proof of Theorem 4.1)

B.2 Proof of Lemma 4.3

Proof of Lemma 4.3: Completeness requires that, for every example set $\mathcal{E}$, if there exists a wrapper satisfying $\mathcal{C}_{\text{HLRT}}$ for each example in $\mathcal{E}$, then $\mathsf{Generalize}_{\text{HLRT}}$ will return such a wrapper. Note that by lines 4.4(h–j), $\mathsf{Generalize}_{\text{HLRT}}$ returns only wrappers that satisfy $\mathcal{C}_{\text{HLRT}}$. Thus to prove the lemma, we need to prove that, if a consistent wrapper exists, then line 4.4(j) is eventually reached.

We prove that $\mathsf{Generalize}_{\text{HLRT}}$ has this property by contradiction. Let $\mathcal{E} = \{\langle P_1, L_1\rangle, \ldots, \langle P_N, L_N\rangle\}$ be a set of examples and w be a wrapper such that $\mathcal{C}_{\text{HLRT}}(w, \mathcal{E})$. Suppose that $\mathsf{Generalize}_{\text{HLRT}}(\mathcal{E})$ never reaches line 4.4(j)—*i.e.*, suppose that the $2K+2$ nested loops iterate completely with $\mathcal{C}_{\text{HLRT}}$ never satisfied.

The fact that $\mathsf{Generalize}_{\text{HLRT}}$ never reaches 4.4(j) implies that it must have neglected to consider wrapper w. In particular, $\mathsf{Generalize}_{\text{HLRT}}$ must have neglected to consider some of the values for one or more of w's HLRT components (h, t, ℓ_1, r_1, *etc.*). We now show that, on the contrary, $\mathsf{Generalize}_{\text{HLRT}}$ considers enough candidates for each HLRT component. Specifically, we show that for each HLRT component, w's value for the component is among the set of candidates. Establishing this fact suffices to prove that w is considered at line 4.4(j), because the algorithm's nested loop control structure (lines 4.4(a–g)) eventually considers every combination of the candidates.

the r_k: Suppose $\mathsf{Generalize}_{\text{HLRT}}$ failed to consider all possible strings for the right-hand delimiter r_k. $\mathsf{Generalize}_{\text{HLRT}}$ does not consider every string as a candidate for each r_k. Rather, lines 4.4(a–b) indicate that the candidates for r_k are the prefixes of the first page's $S_{1,k}$ (the $k^{\underline{\text{th}}}$ intra-tuple separators for the first page's first tuple).[2] But note that this restriction is required in order to satisfy constraint **C1(i)**. Therefore, if the r_k value of wrapper w does not satisfy this restriction, then it will not satisfy **C1**, and therefore w will not satisfy $\mathcal{C}_{\mathrm{HLRT}}(w, \mathcal{E})$. But this is a contradiction, since we assumed that $\mathcal{C}_{\mathrm{HLRT}}(w, \mathcal{E})$ holds.

the ℓ_k $(k > 1)$: A similar argument applies to each left-hand delimiter ℓ_k, for $k > 1$. Lines 4.4(d–e) indicate that the candidates for ℓ_k are the suffixes of the first page's $S_{1,k-1}$ (the $(k-1)^{\underline{\mathrm{th}}}$ intra-tuple separators of the first page's first tuple). But constraint **C2** requires this restriction. Therefore, if the ℓ_k value of wrapper w does not satisfy this restriction, then it will not satisfy **C2**, and therefore w will not satisfy $\mathcal{C}_{\mathrm{HLRT}}(w, \mathcal{E})$.

ℓ_1: A similar argument applies to ℓ_1. Line 4.4(c) indicates that the candidates for ℓ_1 must be suffixes of P_1's head $S_{0,K}$, which is required to satisfy constraint **C3(i)**.

h: A similar argument applies to the head delimiter h: Line 4.4(f) indicates that candidates for h must by substrings of page P_1's head $S_{0,K}$, which is required to satisfy constraint **C3(i)**.

t: A similar argument applies to the tail delimiter t. Line 4.4(g) indicates that candidates for t must by substrings of page P_1's tail $S_{M,K}$, which is required to satisfy constraint **C3(iii)**.

In summary, we have shown that wrapper w is eventually considered in line 4.4(h–i), because sufficient candidates are always considered for each of the $2K+2$ components of w.

□ (Proof of Lemma 4.3)

B.3 Proof of Theorem 4.5

Proof of Theorem 4.5: Recall the discussion of Figure 4.5 on pages 54–55 and pages 60–61. We saw that $\mathsf{Generalize}_{\mathrm{HLRT}}$ and $\mathsf{Generalize}^*_{\mathrm{HLRT}}$ search the same space; they simply traverse it differently. Specifically, $\mathsf{Generalize}_{\mathrm{HLRT}}$ searches the tree completely, while $\mathsf{Generalize}^*_{\mathrm{HLRT}}$ searches the tree greedily (backtracking only over the

[2] See Figure 4.2 and Equation 4.1 for a refresher on the meaning of the partition variables $S_{m,k}$.

bottom three layers of nodes). If we can show that this greedy search never skips a consistent wrapper, then we will have shown that $\mathsf{Generalize}^*_{\mathrm{HLRT}}$ is consistent. (Note that since the search space is the same for both algorithms, we don't need to re-prove that all returned wrappers satisfy $\mathcal{C}_{\mathrm{HLRT}}$.)

Let $\mathcal{E}$ be a set of examples, and suppose that w is an HLRT wrapper that is consistent with $\mathcal{E}$. Without loss of generality, suppose that w is the *only* consistent wrapper—informally, w's leaf in Figure 4.5 is the only one not marked 'X'. By Theorem 4.2, we know that $\mathsf{Generalize}_{\mathrm{HLRT}}$ will return w. We need to show that $\mathsf{Generalize}^*_{\mathrm{HLRT}}$ will too.

(The assumption that w is unique causes no loss of generality because if there were multiple such wrappers, then we could apply this proof technique to each. The only difficulty would be that we can't guarantee which particular wrapper is returned by $\mathsf{Generalize}^*_{\mathrm{HLRT}}$, since we haven't specified the order in which candidates for each component are considered. But no matter which ordering is used, $\mathsf{Generalize}^*_{\mathrm{HLRT}}$ would still be consistent, because consistency requires that *any*—rather than some *particular*—consistent wrapper be returned.)

$\mathsf{Generalize}^*_{\mathrm{HLRT}}$ returns wrapper w iff at each node in the search tree $\mathsf{Generalize}^*_{\mathrm{HLRT}}$ chooses w's value for the corresponding wrapper component. Therefore, $\mathsf{Generalize}^*_{\mathrm{HLRT}}$ will fail to return w if and only if some of w's values are rejected.

the r_k: $\mathsf{Generalize}^*_{\mathrm{HLRT}}$ will not reject w's value for any of the r_k. To see this, note that we assumed w satisfies $\mathcal{C}_{\mathrm{HLRT}}$, and therefore w's value for r_k satisfies **C1**, and therefore $\mathsf{Generalize}^*_{\mathrm{HLRT}}$'s line 4.6(c) will not reject the value.

the ℓ_k $(k > 1)$: $\mathsf{Generalize}^*_{\mathrm{HLRT}}$ will not reject w's value for any of the ℓ_k (for $k > 1$). To see this, note that we assumed w satisfies $\mathcal{C}_{\mathrm{HLRT}}$, and therefore w's value for ℓ_k satisfies **C2**, and therefore $\mathsf{Generalize}^*_{\mathrm{HLRT}}$'s line 4.6(f) will not reject the value.

h, t and ℓ_1: $\mathsf{Generalize}^*_{\mathrm{HLRT}}$ will not reject w's value for h, t and ℓ_1. To see this, note that we assumed w satisfies $\mathcal{C}_{\mathrm{HLRT}}$, and therefore w's value for h, t and

ℓ_1 satisfies **C3**, and therefore $\mathsf{Generalize}^*_{\text{HLRT}}$'s line 4.6(j)—which examines all combinations of these three components—will not reject these values.

□ (Proof of Theorem 4.5)

B.4 Proof of Theorem 4.8

Proof of Theorem 4.8: Suppose $\mathcal{E} = \{\langle P_1, L_1\rangle, \ldots, \langle P_N, L_N\rangle\}$ is a set of N pages drawn independently according to distribution $\mathcal{D}$, and then labeled according to the target wrapper $\mathcal{T}$. Let HLRT wrapper $w = \mathsf{Generalize}_{\text{HLRT}}(\mathcal{E})$. We must show that, if Equation 4.5 holds, then w is PAC—*i.e.*, $E_{\mathcal{T},\mathcal{D}}(w) < \epsilon$ with probability at least $1-\delta$.

$E_{\mathcal{T},\mathcal{D}}(w)$ measures the chance that the hypothesis wrapper w and the target wrapper $\mathcal{T}$ disagree. When do w and $\mathcal{T}$ disagree? That is, under what circumstances is $w(P) \neq \mathcal{T}(P)$, for some particular page P? The consistency constraints **C1**–**C3** capture these circumstances. Specifically, by Definition 4.3 and Theorem 4.1, if any of the constraints **C1**–**C3** fail to hold between w and $\langle P, \mathcal{T}(P)\rangle$, then $w(P) \neq \mathcal{T}(P)$, and thus w and $\mathcal{T}$ disagree about P.

Consider how predicates **C1**–**C3** apply to each of w's components. As indicated in Definition 4.3, these three predicates operate on the different components of w: **C1** constrains each of the K r_k components; **C2** constrains each of the $(K-1)$ ℓ_k components (for $k > 1$); and **C3** constrains h, t, and ℓ_1. Wrapper w disagrees with target $\mathcal{T}$ on page P if any of these $2K$ particular constraints are violated for P. Therefore, the chance of w disagreeing with $\mathcal{T}$ is equal to the chance that one or more of the $2K$ separate constraints are violated. We shall proceed by establishing that, if Equation 4.5 holds then, with high reliability, w and $\mathcal{T}$ disagree only rarely.

Observe that, by the probabilistic union bound[3], we have that the chance of w disagreeing with $\mathcal{T}$ is at most the sum of the chances of the $2K$ individual constraints being violated.

[3] The probabilistic union bound states that, for any events A and B, $\Pr[A \vee B] \leq \Pr[A] + \Pr[B]$.

Suppose we can guarantee (to a particular level of reliability) that the K **C1** constraints and $(K-1)$ **C2** constraints each are violated with probability at most $\frac{\epsilon}{4K-2}$, and that the **C3** constraint is violated with probability at most $\frac{\epsilon}{2}$. If we meet this guarantee, then we will have shown that (with bounded reliability) w and $\mathcal{T}$ disagree with probability at most $K\frac{\epsilon}{4K-2} + (K-1)\frac{\epsilon}{4K-2} + \frac{\epsilon}{2} = \epsilon$ —*i.e.*, we will have shown that w is PAC.

Now consider the following Lemma.

Lemma B.1 (Sample complexity of individual delimiters) *The following statements hold for any target $\mathcal{T}$, distribution $\mathcal{D}$, and $0 < \rho < 1$. (As usual, $\mathcal{E}$ is a set of N examples, comprising a total of M_{tot} tuples, and the shortest example has length R.)*

1. *For any $1 \le k \le K$, if **C1** holds of r_k and every example in $\mathcal{E}$, then, with probability at most $2\left(1-\frac{\rho}{2}\right)^{M_{\text{tot}}}$, **C1** is violated by a fraction ρ or more (with respect to $\mathcal{D}$) of the instances.*
2. *For any $1 < k \le K$, if **C2** holds of ℓ_k and every example in $\mathcal{E}$, then, with probability at most $2\left(1-\frac{\rho}{2}\right)^{M_{\text{tot}}}$, **C2** is violated by a fraction ρ or more (with respect to $\mathcal{D}$) of the instances.*
3. *If **C3** holds between ℓ_1, h, t and every example in $\mathcal{E}$, then, with probability at most $\Phi(R)(1-\rho)^N$, **C3** is violated by a fraction ρ or more (with respect to $\mathcal{D}$) of the instances, where $\Phi(R) = \frac{1}{4}\left(R^{\frac{5}{3}} - 2R^{\frac{4}{3}} + R\right)$.*

Invoking Lemma B.1 with the value $\rho = \frac{\epsilon}{4K-2}$ for the $(2K-1)$ r_k and ℓ_k delimiters, and $\rho = \frac{\epsilon}{2}$ for h, t, and ℓ_1, we have that the chance that $E_{\mathcal{T},\mathcal{D}}(w) > \epsilon$ is at most

$$2K\left(1-\frac{\epsilon}{4K-2}\right)^{M_{\text{tot}}} \; + \; 2(K-1)\left(1-\frac{\epsilon}{4K-2}\right)^{M_{\text{tot}}} \; + \; \Phi(R)\left(1-\frac{\epsilon}{2}\right)^{N}$$

and therefore ensuring that

$$(4K-2)\left(1-\frac{\epsilon}{4K-2}\right)^{M_{\text{tot}}} \; + \; \Phi(R)\left(1-\frac{\epsilon}{2}\right)^{N} \quad < \quad \delta$$

suffices to ensure that w is PAC, as claimed. (For clarity, we can rewrite this equation as

$$\Psi(K)\left(1-\frac{\epsilon}{\Psi(K)}\right)^{M_{\mathrm{tot}}} \;+\; \Phi(R)\left(1-\frac{\epsilon}{2}\right)^{N} \;<\; \delta$$

where $\Psi(K) = 4K - 2$.) To complete the proof of Theorem 4.8, it suffices to establish Lemma B.1.

□ (Proof of Theorem 4.8)

Proof of Lemma B.1: We derive each part of the Lemma in turn.

Part 1 (ℓ_k). Wrapper component ℓ_k is learned by examining the M_{tot} tuples in $\mathcal{E}$. We are looking for some string ℓ_k that occurs just to the left of the instances of the $k^{\underline{\mathrm{th}}}$ attribute.

More abstractly, we can treat the task of learning ℓ_k as one of identifying a common proper suffix[4] of a set of strings $\{s_1, \ldots, s_{M_{\mathrm{tot}}}\}$, where the s_i are the values of the $S_{m,k-1}$ partition variables. For example, when learning ℓ_2 using the example shown in Figure 2.1(c), the set of examples would be the four strings

{`</B> <I>`, `</B> <I>`, `</B> <I>`, `</B> <I>`}.

Note that these four strings are the values of $S_{1,1}$, $S_{2,1}$, $S_{3,1}$ and $S_{4,1}$ (which in this example happen to be identical).

Given this description of the learning task, we need to determine the chance that ℓ_k is not a proper suffix of at least a fraction ρ (with respect to $\mathcal{D}$) of the instances. (We can treat $\mathcal{D}$ as a distribution over the s_i even though $\mathcal{D}$ is in fact a distribution over pages, because, from the perspective of learning a single ℓ_k, an example page is equivalent to the examples of ℓ_k it provides.)

[4] A proper suffix is a suffix that occurs *only* as a suffix; see Appendix C for a description of our string algebra.

Let s be a string. Notice that each suffix of s can be represented as an integer I, where I is the index in s of the suffix's first character. For example, the suffix abc of the strings abcpqrabc is identified with the integer $I = 7$, because abcpqrabc$[7] =$ abc. Since a suffix and its integral representation are equivalent, we use the two terms interchangeably in this proof.

The set of all *proper* suffixes of a string s can be represented as an interval $[1, U]$, where U indicates s's shortest proper suffix. For example, the set of proper suffixes of the string abcpqrabc can be represented as the interval $[1, 6]$, because abc (which starts at position 7) is the string's longest improper suffix. Note that every integer in $[1, U]$ corresponds to a proper suffix, and every proper suffix corresponds to an integer in $[1, U]$: every string is a proper suffix of itself, and if some particular suffix is improper, then all shorter suffixes are also improper.

In addition to being a proper suffix, ℓ_k must be a *common* proper suffix of each example. The set of common proper suffixes of a set of strings $\{s_1, \ldots\}$ is captured by the interval $[L, U]$. L is the index into s_1 indicating the longest common suffix, while U is the index into s_1 indicating the shortest common proper suffix. Notice that implicitly these indices use s_1 as a reference string.

Observe that every integer in $[L, U]$ corresponds to a common proper suffix, and every common proper suffix corresponds to an integer in $[L, U]$: by construction, suffixes corresponding to integers $\geq L$ are common suffixes, while those corresponding to integers $\leq U$ are proper suffixes. Therefore, the intersection of these two intervals corresponds exactly to the set of common proper suffixes.

Figure B.1 uses a suffix-tree representation to illustrate that $[5, 6]$ represents the common proper suffixes of the strings $s_1 =$ rqpabcd, $s_2 =$ dzqpabcd, $s_3 =$ dqzabcd, $s_4 =$ pqzabcd and $s_5 =$ qpabcd. That is, bcd and cd are the only proper suffixes that are common to all five examples.

Figure B.2 illustrates the point of the preceding discussion: learning a common proper suffix corresponds to learning the integer I based on a set of observed upper

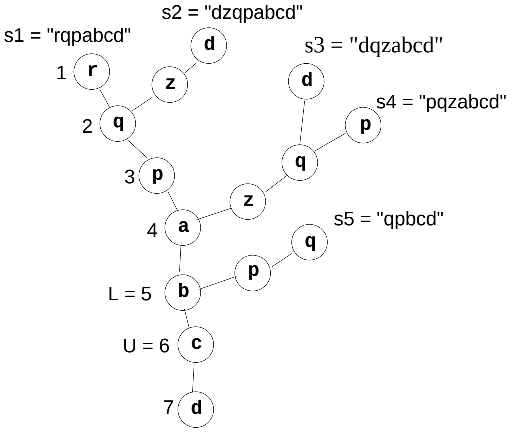


Figure B.1: *A suffix tree showing that the common proper suffixes of the five examples can be represented as the interval* $[L, U] = [5, 6]$.

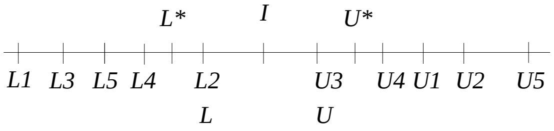


Figure B.2: *An example of learning the integer* I *from lower bounds* $\{L_1, \ldots, L_5\}$ *and upper bounds* $\{U_1, \ldots, U_5\}$.

and lower bounds on I, where I is the index into the reference string s_1 representing the target value for ℓ_k. Each example s_i provides a lower bound L_i for I as well as an upper bound U_i for I. More formally, L_i is the index in s_1 of the longest suffix shared by s_i and s_1, and U_i is the index into s_1 of the shortest proper suffix of s_i. Given a set of examples, our learner outputs an hypothesis corresponding to integers in the interval $[L, U]$, where $L = \max_i L_i$ is the greatest lower bound, and $U = \min_i U_i$ is the least upper bound.

At this point, we can analyze the learning task in terms of a PAC model. As shown in Figure B.2, define $L^* < I$ to be the integer such that exactly a fraction $\frac{\varrho}{2}$ of the probability mass of $\mathcal{D}$ corresponds to strings providing a lower bound in the range $[L^*, I]$. Conceptually, L^* is determined by starting at I and moving to the left until the strings depositing their lower-bound L_i within $[L^*, I]$ collectively have probability mass exactly $\frac{\rho}{2}$. Similarly, define U^* to be the integer such that $[I, U^*]$ has mass exactly $\frac{\rho}{2}$.[5]

The learner will output some integer from $[L, U]$; what is the chance that a randomly selected string will disagree with this choice? By construction, if $L < L^*$ or $U > U^*$, then this chance will be at least $2\frac{\rho}{2} = \rho$. What is the chance that $L < L^*$ or $U > U^*$? By the union bound, we have that the chance of this disjunctive event is at most the sum of the probabilities of each disjunct. Therefore, by calculating this sum, we can obtain the probability that the learned hypothesis has error more than ρ.

What is the chance that $L < L^*$? $L < L^*$ if, amongst the M_{tot} example strings, we have seen none such that $L_i \geq L^*$ (since, if we had seen such a string, then we would have $L \geq L^*$). By the construction of L^*, the chance that a single string's lower bound is less than L^* is $1 - \frac{\rho}{2}$. So the chance that we see no such examples is $(1 - \frac{\rho}{2})^{M_{\text{tot}}}$.

A symmetrical argument applies to the chance that $U > U^*$: the chance that $U > U^*$ is exactly $(1 - \frac{\rho}{2})^{M_{\text{tot}}}$.

Summing these two bounds, we have that the chance that ℓ_k disagrees with a fraction ρ or more of the instances is at most $2\left(1 - \frac{\rho}{2}\right)^{M_{\text{tot}}}$.

Part 2 (r_k). A similar proof technique can be used to show that the same bound applies to learning r_k. The difference is that the observed lower bounds and upper bounds are generated from the examples somewhat differently.

[5] Since the interval is taken to be integral, it may be that no such L^* or U^* exists. For example, if the interval $[3, I]$ has mass $< \frac{\rho}{2}$ while $[4, I]$ has mass $> \frac{\rho}{2}$, then L^* is not defined. In this case we simply choose $L^* = 3.5$. More generally, we allow L^* and U^* to be non-integral if needed. This technicality does not affect the rest of the proof.

Specifically, r_k is learned from a set of $M_{\rm tot}$ pairs of examples strings

$$\{\langle s_1^a, s_1^b\rangle, \ldots, \langle s_i^a, s_i^b\rangle, \ldots, \langle s_{M_{\rm tot}}^a, s_{M_{\rm tot}}^b\rangle\}.$$

The s_i^a are $A_{m,k}$ partition variables, and the s_i^b are the $S_{m,k}$ partition variables. For example, when learning r_1, the country/code example page provides the four pairs of strings

$$\{\langle \texttt{Congo, <B> </I>}\rangle,\ \langle \texttt{Egypt, <B> </I>}\rangle,\ \langle \texttt{Belize, <B> </I>}\rangle,\ \langle \texttt{Spain, <B> </I>}\rangle\}.$$

Constraint **C1** specifies that r_k must be a prefix of every s_i^b and that r_k must not occur in any s_i^a. We need to determine the chance that r_k does not satisfy **C1** for a fraction ρ or more of the instances.

An hypothesis is represented as an index I into the first example. For example, the prefix `abc` of the example $\langle \texttt{pqrs}, \texttt{abcdef}\rangle$ would be represented by the integer $I = 3$, since $\texttt{abcdef}[1, 3] = \texttt{abc}$.

The interval $[L, U]$ represents all the hypothesis satisfying constraint **C1**. As with part 1, each example corresponds to a value L_i and U_i, and the learning task involves returning any hypothesis in $[L, U]$, where $L = \max_i L_i$ and $U = \min_i U_i$.

Specifically, L_i for example $\langle s_i^a, s_i^b\rangle$ is the shortest prefix of s_1^a satisfying constraint **C1**. U_i is the longest such prefix. For instance, if we are trying to learn r_k from the following pairs of strings:

$$\left\{\langle s_1^a = \texttt{abc},\ s_1^b = \texttt{abcdef}\rangle,\ \langle s_2^a = \texttt{pqr},\ s_2^b = \texttt{abcdex}\rangle\right\}$$

then we would translate these examples into $L_1 = 4$, $U_1 = 6$, $L_2 = 4$ and $U_2 = 5$, so that $L = 4$ and $U = 5$.

The key observation about this construction is the interval $[L, U]$ is equivalent to the set of hypothesis satisfying **C1**: $G \in [L, U] \iff G$ satisfies **C1**. (Proof of ⇒: Suppose G did not satisfy **C1** for example i; thus $L_i > G$ or $U_i < G$; therefore $L > G$ or $U < G$; but $G \in [L, U]$—contradiction. Proof of ⇐: Suppose $G \notin [L, U]$; then there exists an example i such that $L_i > G$ or $U_i < G$; therefore G does not satisfy **C1** for example i—contradiction.)

From this point, the proof of part 2 is identical to the proof of part 1: the chance that r_k disagrees with a fraction ρ or more of the instances is at most $2\left(1-\frac{\rho}{2}\right)^{M_{\text{tot}}}$.

Part 3 (h, t, ℓ_1). We can think of finding values for h, t, and ℓ_1 as a problem of learning within a "restricted" hypothesis space $\mathcal{H}_{h,t,\ell_1}$. Given the first example page P_1, there are only a finite number of choices for h, t, ℓ_1, and thus $\mathcal{H}_{h,t,\ell_1}$ has finite cardinality. Specifically, Equation 4.4 states that the number of combinations of h, t, and ℓ_1 is bounded by $\mathrm{O}(R^{\frac{5}{3}})$, where $R = \min_n |P_n|$ is the length of the shortest example page.[6]

How many hypotheses exactly? In the analysis leading to Equation 4.4, we were concerned only with asymptotic complexity. We can re-examine the relevant parts of $\mathsf{Generalize}^*_{\text{HLRT}}$ to determine the actual number of h, t, ℓ_1 combinations, $|\mathcal{H}_{h,t,\ell_1}|$. Note that $|\mathcal{H}_{h,t,\ell_1}|$ is a function (only) of R; let $|\mathcal{H}_{h,t,\ell_1}| = \Phi(R)$. Then lines 4.6(g–i) reveal that:

$$\begin{aligned}\Phi(R) &= \underbrace{\frac{\sqrt[3]{R}\left(\sqrt[3]{R}-1\right)}{2}}_{\substack{h\text{ is a}\\ \text{substring of}\\ \text{a string of}\\ \text{length }\sqrt[3]{R}}} \times \underbrace{\frac{\sqrt[3]{R}\left(\sqrt[3]{R}-1\right)}{2}}_{\substack{t\text{ is a}\\ \text{substring of}\\ \text{a string of}\\ \text{length }\sqrt[3]{R}}} \times \underbrace{\sqrt[3]{R}}_{\substack{\ell_1\text{ is a}\\ \text{suffix of a}\\ \text{string of}\\ \text{length}\\ \sqrt[3]{R}}}\\ &= \frac{1}{4}\left(R^{\frac{5}{3}} - 2R^{\frac{4}{3}} + R\right).\end{aligned}$$

To summarize, we have determined $\Phi(R)$, which is the number of hypotheses $|\mathcal{H}_{h,t,\ell_1}|$ examined by the $\mathsf{Generalize}^*_{\text{HLRT}}$ algorithm.

PAC bounds for finite hypothesis spaces are well known in the literature (*e.g.*, [Russell & Norvig 95, pp 553–5]). Specifically, it is straightforward to show that, if

[6] This proof requires Assumption 4.1 only to obtain this bound; if the assumption doesn't hold, then we get a looser, though still correct, model.

some hypothesis from class $\mathcal{H}$ is consistent with N examples, then the chance that it disagrees with at least a fraction ρ of the instances is at most $|\mathcal{H}|(1-\rho)^N$.

Applying this bound to the problem of learning $\mathcal{H}_{h,t,\ell_1}$, we have that the chance that h, t, and ℓ_1 jointly disagree with at least a fraction ρ of the instances is at most $\Phi(R)(1-\rho)^N$.

□ (Proof of Lemma B.1)

B.5 Proof of Theorem 5.1 (Details)

Proof of Theorem 5.1 (Details):

Assertion 1. Figure B.3 lists nine pages. For each page, the label (not explicitly listed) extracts the information $\{\langle \texttt{A11}, \texttt{A12}\rangle, \langle \texttt{A21}, \texttt{A22}\rangle, \langle \texttt{A31}, \texttt{A32}\rangle\}$ from the corresponding page. For example, the page marked *(A)*, `pA11qA12rA21sA22tA31uA32v`, is labeled $\{\langle\langle 2,4\rangle,\langle 6,8\rangle\rangle,\langle\langle 10,12\rangle,\langle 14,16\rangle\rangle,\langle\langle 18,20\rangle,\langle 22,24\rangle\rangle\}$.[7]

Figure B.3 lists each of the wrapper classes: LR, HLRT, OCLR and HOCLRT. The symbol "×" in a particular page's row and a particular class's column indicates that no wrapper in the class is consistent with the given page. For example, the symbol "×" at page *(A)* under column HLRT indicates that no HLRT wrapper exists which can extract the given content from page *(A)*. On the other hand, the HLRT wrapper $\langle \texttt{h}, \texttt{t}, \texttt{[}, \texttt{]}, \texttt{(}, \texttt{)}\rangle$ for page *(C)* indicates that the given wrapper can indeed extracted the desired content from the page.

We must verify each cell of the nine-by-four matrix. The cells *not* marked by "×" can be easily verified by simply running the corresponding wrapper execution. For instance, to verify that the cell "*(C)*–HLRT" should be marked $\langle \texttt{h}, \texttt{t}, \texttt{[}, \texttt{]}, \texttt{(}, \texttt{)}\rangle$, we simply run ExecHLRT with arguments `-[h[A11](A12)[A21](A22)[A31](A32)t` and $\langle \texttt{h}, \texttt{t}, \texttt{[}, \texttt{]}, \texttt{(}, \texttt{)}\rangle$ and then verify that the output is correct.

[7] Recall our warning in Footnote 1 on page 77 when examining Figures B.3 and B.4.

Verifying the cells marked "×" is more difficult. Each requires a detailed argument specific to the particular page and wrapper class in question. In this proof, we provide such arguments for just two cells.[8]

- Consider row *(I)*. The page `[ho[A11](A12)cox[A21](A22)co[A31](A32)c` is claimed to be wrappable by OCLR and HOCLRT, but not LR or HLRT. To see that no consistent LR wrapper exists, note that ℓ_1 must be a common proper suffix of the three strings `[ho[`, `)cox[` and `)co[`. Clearly, `[` is the only such string. But using this ℓ_1 candidate in an LR wrapper causes ExecLR to get confused by the page's head `[ho[`. Specifically, when attempting to extract the page's first attribute (`A11`), ExecLR will extract `[ho[A11` instead, because ℓ_1 occurs first at position 1 instead of position 5.

  To see that `[ho[A11](A12)cox[A21](A22)co[A31](A32)c` can not be wrapped by HLRT, consider the tail delimiter t. t must be a substring of the tail `)c`. However, we can not use any of the three substrings (`)c`, `)` and `c`), because all three occur between the tuples and therefore confuse ExecHLRT.

- As a second example, consider row *(C)*, `-[h[A11](A12)[A21](A22)[A31](A32)t`. As with *(I)*, there is no consistent LR wrapper because all candidates for ℓ_1 confuse ExecLR.

  To see that no OCLR wrapper can handle `-[h[A11](A12)[A21](A22)[A31](A32)t`, notice that all candidates for the opening delimiter o confuse ExecOCLR.

Finally, notice that row *(A)* in Figure B.3 corresponds to region *(A)* in Figure 5.1, *(B)* in Figure B.3 corresponds to region *(B)* in Figure 5.1, and so on. Thus we

[8] We also implemented a computer program to enumerate and rule out every possible wrapper in the given class. The program is complete, in that it will always find a wrapper if one exists, and it will always terminate, so that if no wrapper exists, then we can in fact use the program to verify this fact.

have exhibited one pair $\langle P, L\rangle$ in each of the regions *(A)*, …, *(I)*, which establishes the first assertion in the proof of Theorem 5.1.

Assertions 2 and 3. Lemmas B.2 and B.3 below establish the second and third assertions, respectively.

□ (Proof of Theorem 5.1 (Details))

Lemma B.2 $\Pi(\textsc{lr}) \subset \Pi(\textsc{oclr})$.

Proof of Lemma B.2: We need to show that, for every pair $\langle P, L\rangle \in \Pi(\textsc{lr})$, if LR wrapper $w = \langle \ell_1, r_1, \ldots, \ell_K, r_K\rangle$ satisfies $w(P) = L$, then OCLR wrapper $w' = \langle \ell_1, \phi, \ell_1, r_1, \ldots, \ell_K, r_K\rangle$ satisfies $w'(P) = L$.[9] If we establish this assertion, then we will have established that $\Pi(\textsc{lr}) \subset \Pi(\textsc{oclr})$.

To see that this assertion holds, note that OCLR wrapper w''s o component has the value ℓ_1, while c is the empty string ϕ. But under these circumstances, the ExecOCLR procedure is equivalent to the ExecLR procedure. Specifically, when $o = \ell_1$, lines (a–b) of ExecOCLR are equivalent to line (a–b) of ExecLR. Furthermore, when $c = \phi$, line (c) of ExecOCLR is a “no-op”.

Since we know $\mathsf{ExecLR}(w, P) = L$, we conclude $\mathsf{ExecOCLR}(w', P) = L$.

□ (Proof of Lemma B.2)

Lemma B.3 $\Pi(\textsc{hlrt}) \subset \Pi(\textsc{hoclrt})$.

Proof of Lemma B.3: Substantially the same proof technique applies to Lemma B.3 and Lemma B.2. In this case, when $o = \ell_1$, then lines (a–b) of ExecHOCLRT are equivalent to lines 4.1(b–c) of ExecHLRT, and when $c = \phi$, then line (c) of ExecHOCLRT is a “no-op”.

□ (Proof of Lemma B.3)

[9] Recall that the empty string ϕ is defined in Appendix C.

		LR	HLRT	OCLR	HOCLRT
region	example	$\langle \ell_1, r_1, \ell_2, r_2 \rangle$	$\langle h, t, \ell_1, r_1, \ell_2, r_2 \rangle$	$\langle o, c, \ell_1, r_1, \ell_2, r_2 \rangle$	$\langle h, t, o, c, \ell_1, r_1, \ell_2, r_2 \rangle$
(A)	`pA11qA12rA21sA22tA31uA32v`	×	×	×	×
(B)	`o[ho[A11](A12)cox[A21](A22)co[A31](A32)c`	×	×	×	⟨h,),o,c,[,],(,)⟩
(C)	`-[h[A11](A12)[A21](A22)[A31](A32)t`	×	⟨h,t,[,],(,)⟩	×	⟨h,t,[,),[,],(,)⟩
(D)	`[h[A11](A12)h-[A21](A22)h[A21](A22)t`	×	⟨h,t,[,],(,)⟩	⟨h,),[,],(,)⟩	⟨h,t,x,),[,],(,)⟩
(E)	`ho[A11](A12)co[A21](A22)co[A31](A32)ct`	⟨[,],(,)⟩	⟨h,t,[,],(,)⟩	⟨o,c,[,],(,)⟩	⟨h,t,o,c,[,],(,)⟩
(F)	`ho[A11](A12)cox[A21](A22)co[A31](A32)c`	⟨[,],(,)⟩	×	⟨o,c,[,],(,)⟩	⟨h,),o,c,[,],(,)⟩
(G)	`[A11](A12)t[A21](A22)[A31](A32)t`	⟨[,],(,)⟩	×	⟨[,),[,],(,)⟩	×
(H)	`x[o[A11](A12)o[A21](A22)ox[A31](A32)`	×	×	⟨o,),[,],(,)⟩	×
(I)	`[ho[A11](A12)cox[A21](A22)co[A31](A32)c`	×	×	⟨o,c,[,],(,)⟩	⟨h,),o,c,[,],(,)⟩

Figure B.3: *One point in each of the regions in Figure 5.1.*

B.6 Proof of Theorem 5.10 (Details)

Proof of Theorem 5.10 (Details):

Assertion 1. The proof technique used for Theorem 5.1 can be applied to the present theorem. Figure B.4 lists one point in each of the regions marked *(J)* through *(U)* in Figure 5.3.

Assertions 2 and 3. Lemmas B.4 and B.5 below establish the second and third assertions, respectively.

□ (Proof of Theorem 5.10 (Details))

Lemma B.4 $(\Pi(\text{N-LR}) \cap \Pi(\text{HLRT})) \subset \Pi(\text{LR})$.

Proof of Lemma B.4: Let $\langle P, L\rangle$ be some page wrappable by both N-LR and HLRT: $\langle P, L\rangle \in (\Pi(\text{N-LR}) \cap \Pi(\text{HLRT}))$. Since $\langle P, L\rangle$ is wrappable by HLRT, it must have a tabular (as opposed to nested) structure. Let w be an N-LR wrapper consistent with $\langle P, L\rangle$. Since P has a tabular structure, then $\mathsf{ExecLR}(w, P) = L$; *i.e.*, when treated as an LR wrapper, the N-LR wrapper w is consistent with $\langle P, L\rangle$. (To see this, note that P is tabular and so line (i) of ExecN-LR always finds attribute $k+1$ after attribute k. Therefore, the ExecLR routine operates properly when using the same attribute ℓ_k.) Since there exists a w such $\mathsf{ExecLR}(w, P) = L$, we have that $\langle P, L\rangle \in \Pi(\text{LR})$.

□ (Proof of Lemma B.4)

Lemma B.5 $(\Pi(\text{N-HLRT}) \cap \Pi(\text{LR})) \subset \Pi(\text{HLRT})$.

Proof of Lemma B.5: Essentially the same proof applies at for Lemma B.4. The difference is that instead of treating an N-LR wrapper as an LR wrapper (the technique used in the previous proof), we rely on the fact that if a given page is tabular, then an N-HLRT wrapper operates correctly when treated as an HLRT wrapper.

		LR	HLRT	N-LR	N-HLRT
region	example	$\langle \ell_1, r_1, \ell_2, r_2 \rangle$	$\langle h, t, \ell_1, r_1, \ell_2, r_2 \rangle$	$\langle \ell_1, r_1, \ell_2, r_2 \rangle$	$\langle h, t, \ell_1, r_1, \ell_2, r_2 \rangle$
(J)	`pA11qA12rA21sA22tA31uA32v`	×	×	×	×
(K)	`[h[A11][(A12)[A21](A22)[A31](A32)t`	×	⟨h, t, [,], (,)⟩	×	×
(L)	`[A11][(A12)t[A21](A22)[A31](A32)t`	⟨[,], (,)⟩	×	×	×
(M)	`h[A11][(A12)[A21](A22)[A31](A32)t`	⟨[,], (,)⟩	⟨h, t, [,], (,)⟩	×	×
(N)	`h[A11]-(A12)[A21](A22)[A31](A32)t(`	⟨[,], (,)⟩	⟨h, t, [,], (,)⟩	×	⟨h, t, [,], (,)⟩
(O)	`[h[A11](A12)[A21](A22)[A31](A32)t`	×	⟨h, t, [,], (,)⟩	×	⟨h, t, [,], (,)⟩
(P)	`[h[A11i][A11ii](A12)[A21](A22)[A31](A32)t`	×	×	×	⟨h, t, [,], (,)⟩
(Q)	`[A11i][A11ii](A12)t[A21](A22)[A31](A32)t`	×	×	⟨[,], (,)⟩	×
(R)	`[A11](A12)t[A21](A22)[A31](A32)t`	⟨[,], (,)⟩	×	⟨[,], (,)⟩	×
(S)	`h[A11])t(A12)[A21](A22)[A31](A32)t`	⟨[,], (,)⟩	⟨h, t, [,], (,)⟩	⟨[,], (,)⟩	×
(T)	`h[A11](A12)[A21](A22)[A31](A32)t`	⟨[,], (,)⟩	⟨h, t, [,], (,)⟩	⟨[,], (,)⟩	⟨h, t, [,], (,)⟩
(U)	`h[A11i][A11ii](A12)[A21](A22)[A31](A32)t`	×	×	⟨[,], (,)⟩	⟨h, t, [,], (,)⟩

Figure B.4: *One point in each of the regions in Figure 5.3.*

□ (Proof of Lemma B.5)

B.7 Proof of Theorem 6.1

Proof of Theorem 6.1: Let Δ be a recognizer library satisfying Assumption 6.1, let P be a page, let L^* be P's true label, and let $\mathcal{L} = \mathsf{Corrob}(P, \Delta)$. Correctness requires that there exists some member of $\mathcal{L}$ that matches L^*. To establish this assertion, we first show how to construct a label L that matches L^*, and then show that $L \in \mathcal{L}$.

Constructing L from L^*. L's attribute ordering is identical to that of L^*. L also contains exactly the same instances as L^*, except that any instance $\langle b, e, F_k \rangle$ from L^* is replaced by "?" if $\langle b, e \rangle$ was not recognized (*i.e.*, $\langle b, e \rangle \notin \mathcal{R}_{F_k}(P)$).

For example, consider the following example recognizer library output:

$\mathcal{R}_{\text{CAP}}$ *perfect*
$\langle 29, 34 \rangle$
$\langle 49, 54 \rangle$
$\langle 69, 74 \rangle$

$\mathcal{R}_{\text{CTRY}}$ *incomplete*
$\langle 35, 40 \rangle$

$\mathcal{R}_{\text{CODE}}$ *perfect*
$\langle 22, 27 \rangle$
$\langle 42, 47 \rangle$
$\langle 62, 67 \rangle$

In this case, we construct L from L^* as follows:

true label L^*

CTRY	CODE	CAP
$\langle 15, 20 \rangle$	$\langle 22, 27 \rangle$	$\langle 29, 34 \rangle$
$\langle 35, 40 \rangle$	$\langle 42, 47 \rangle$	$\langle 49, 54 \rangle$
$\langle 55, 60 \rangle$	$\langle 62, 67 \rangle$	$\langle 69, 74 \rangle$

$\Longrightarrow$

constructed label L

CTRY	CODE	CAP
?	$\langle 22, 27 \rangle$	$\langle 29, 34 \rangle$
$\langle 35, 40 \rangle$	$\langle 42, 47 \rangle$	$\langle 49, 54 \rangle$
?	$\langle 62, 67 \rangle$	$\langle 69, 74 \rangle$

Note that L is identical to L^* except that two CTRY instances—$\langle 15, 20 \rangle$ and $\langle 55, 60 \rangle$—are omitted, because neither is a member of $\mathcal{R}_{\text{CTRY}}(P)$.

Showing that $L \in \mathcal{L}$. It is clear from the previous discussion that L matches L^*, in the sense required for the set $\mathcal{L}$ to qualify as a solution to the labeling problem $\langle P, \Delta \rangle$. Thus to complete the proof we need to show that $L \in \mathcal{L}$.

Let "$\prec$" be the attribute ordering of L^* and L; note that the ordering is the same for both. Let $\{\ldots, \langle b, e, F_k\rangle, \ldots\}$ be the instances in L. Now partition $\{\ldots, \langle b, e, F_k\rangle, \ldots\}$ into two parts: $I_{\mathrm{UNSND}} = \{\langle b, e, F_k\rangle \mid \textsc{unsnd}(\mathcal{R}_{F_k})\}$, the set of instances recognized by the unsound recognizers; and $I_{\neg\mathrm{UNSND}} = \{\langle b, e, F_k\rangle \mid \neg\textsc{unsnd}(\mathcal{R}_{F_k})\}$, the set of instances recognized by the incomplete or perfect recognizers.

To show that $L \in \mathcal{L}$, we must demonstrate that the following five assertions hold.

1. $I_{\neg\mathrm{UNSND}}$ is the set of instances returned by the call to NTPSet at line 6.1(a), so that $\mathsf{NTPSet} = I_{\neg\mathrm{UNSND}}$.

2. I_{UNSND} is one of the sets of instances returned by the call to $\mathsf{PTPSets}$ at line 6.1(b); so that eventually $\mathsf{PTPSet} = I_{\mathrm{UNSND}}$;

3. "$\prec$" is one of the orderings returned by the call to Orders at line 6.1(c).

4. When invoked at line 6.1(d) with $\mathsf{NTPSet} = I_{\neg\mathrm{UNSND}}$, $\mathsf{PTPSet} = I_{\mathrm{UNSND}}$ and attribute ordering "$\prec$", the function invocation $\mathsf{Consistent?}(\text{“}\prec\text{”}, \mathsf{NTPSet} \cup \mathsf{PTPSet})$ returns $\textsc{true}$, thereby ensuring that:
$$\mathsf{MakeLabel}(\text{“}\prec\text{”}, \mathsf{NTPSet} \cup \mathsf{PTPSet}) \quad \in \quad \mathcal{L}.$$

5. Finally, when invoked at line 6.1(e) with $\mathsf{NTPSet} = I_{\neg\mathrm{UNSND}}$, $\mathsf{PTPSet} = I_{\mathrm{UNSND}}$ and attribute ordering "$\prec$", label L is returned by the function invocation
$$\mathsf{MakeLabel}(\text{“}\prec\text{”}, \mathsf{NTPSet} \cup \mathsf{PTPSet}).$$

We now establish each item on this list.

1. The function NTPSet simply collect the instances recognized by each of the perfect or incomplete recognizers. So we have that $\langle b, e, F_k\rangle \in I_{\neg\mathrm{UNSND}} \iff \langle b, e, F_k\rangle \in \mathsf{NTPSet}(P, \Delta)$, and thus $I_{\neg\mathrm{UNSND}} = \mathsf{NTPSet}(P, \Delta)$.

2. Consider partitioning I_{UNSND} according to the attribute F_k to which they belong: $I_{\text{UNSND}}^{F_k} = \{\langle b, e\rangle \mid \langle b, e, F_k\rangle \in I_{\text{UNSND}}\}$. Note that unsound recognizers never produce false negatives, and therefore $I_{\text{UNSND}}^{F_k} \subseteq \mathcal{R}_{F_k}(P)$. Furthermore, note that $|I_{\text{UNSND}}^{F_k}| = M$, where M is the number of rows in L^*. Now, since the PTPSets subroutine returns the cross product of *all* subsets of size M for *all* sets $\mathcal{R}_{F_k}(P)$ of instances recognized by unsound recognizers, we know that the particular set I_{UNSND} will be among those returned.

3. Orders returns the set of all possible attribute orderings and therefore it must include the ordering "$\prec$".

4. To review, we know now that L's attribute ordering "$\prec$", NTP instances $I_{\neg\text{UNSND}}$, and PTP instances I_{UNSND} will eventually be considered at lines 6.1(d–e). Thus we know that Corrob is prepared to add L to $\mathcal{L}$. So the only remaining questions are: Does Consistent? return TRUE, and, if so, does MakeLabel correctly produce L from these inputs?

   To see that Consistent? returns TRUE for these inputs, note that Consistent? verifies that a label exists with the given attribute ordering and all the given instances. Since L is such a label, we know that Consistent? must return TRUE.

5. Moreover, note that there is *exactly one* such label—*i.e.*, L is the *only* label that can be constructed from "$\prec$" and $I_{\neg\text{UNSND}} \cup I_{\text{UNSND}}$. To see this, note that $I_{\neg\text{UNSND}} \cup I_{\text{UNSND}}$ contain a subset of L^*'s instances, and since the attribute ordering is fixed, each instance recognized by an incomplete recognizer belongs to exactly one row in L. Therefore, MakeLabel returns L for the given inputs.

□ (Proof of Theorem 6.1)

Appendix C

STRING ALGEBRA

The consistency predicates **C1–C5** and the $\mathsf{Exec}\mathcal{W}$ execution functions for each wrapper class $\mathcal{W}$ are defined in terms of the following concise string algebra.

- We assume an alphabet Σ. We don't restrict the alphabet, though we have in mind the ASCII characters. The symbol "$\Downarrow$" refers to the new-line character.

- In this Appendix, s, s', *etc.* are strings over Σ. The length of string s is written $|s|$. The empty string is denoted "ϕ". Adjacency indicates concatenation: "$s\,s'$" denotes the string s followed by s'.

- "/" is the string *search* operator: string s/s' is the suffix of s starting from the first occurrence of s'. For example, $\texttt{abcdecdf}/\texttt{cd} = \texttt{cdecdf}$. The empty string ϕ occurs at the beginning of every: $s/\phi = s$.

- If s does not contain s', then we write $s/s' = \Diamond$. For example, $\texttt{abc}/\texttt{xyz} = \Diamond$. We stipulate that "/" propogates failure: $s/\Diamond = \Diamond/s = \Diamond$. Also, for convenience we define $|\Diamond| = \infty$.

- One common use of the search operator is to determine if one string is a proper suffix of another. We say that s' is a *proper suffix* of string s if s' is a suffix of s and moreover s' occurs *only* as a suffix of s. Note that s' is a proper suffix of s if and only if $s/s' = s'$.

- A second common use of the string search operator is to determine if one string occurs before another in some third string. Note that s' occurs after s'' in s if and only if $|s/s'| < |s/s''|$.

- "#" is the string *index* operator: $s\#s' = |s| - |s/s'| + 1$. As with s/s', $s\#s' = \Diamond$ if s does not contain s'. For example, $\texttt{abcdefg}\#\texttt{cde} = 3$, while $\texttt{abcdefg}\#\texttt{xyz} = \Diamond$.

- The *substring* operator $s[b, e]$ extracts from s characters b though e, where b and e are one-based integral indices into s. For example, $\texttt{abcdef}[3, 4] = \texttt{cd}$. $s[b]$ is an abbreviation for $s[b, |s|]$. For example, $\texttt{abcdef}[3] = \texttt{cdef}$.

BIBLIOGRAPHY

[Adali et al. 96] Adali, S., Candan, K., Papakonstantinou, Y., and Subrahmanian, V. Query caching and optimization in distributed mediator systems. In *Procceedings of SIGMOD-96*, 1996.

[Aiken 95] Aiken, P. *Data Reverse Engineering: Staying the Legacy Dragon.* McGraw Hill, 1995.

[Andreoli et al. 96] Andreoli, J., Borghoff, U., and Pareschi, R. The Constraint-Based Knowledge Broker Mode: Semantics, Implementation and Analysis. *J. Symbolic Computation*, 21(4):635–67, 1996.

[Angluin & Laird 88] Angluin, D. and Laird, P. Learning from noisy examples. *Machine Learning*, 2(4):343–70, 1988.

[Angluin & Smith 83] Angluin, D. and Smith, C. Inductive inference: Theory and methods. *ACM Computing Surveys*, 15:237–69, 1983.

[Angluin 82] Angluin, D. Inference of reversible languages. *J. ACM*, 29(3):741–65, 1982.

[Angluin 87] Angluin, D. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.

[Angluin 92] Angluin, D. Computational learning theory: survey and selected bibliography. In *Proc. 24th ACM Symp. Theory Comp.*, pages 351–69, 1992.

[ANSI 92] ANSI. Database Language SQL, 1992. Standard X3.135-1992.

[Arens et al. 96] Arens, Y., Knoblock, C., Chee, C., and Hsu, C. SIMS: Single interface to multiple sources. TR RL-TR-96-118, USC Rome Labs, 1996.

[ARPA 95] ARPA. *Proc. 6th Message Understanding Conf.* Morgan Kaufmann, 1995.

[Ashish & Knoblock 97a] Ashish, N. and Knoblock, C. Semi-automatic wrapper generation for Internet information sources. In *Proc. Cooperative Information Systems*, 1997.

[Ashish & Knoblock 97b] Ashish, N. and Knoblock, C. Wrapper Generation for Semi-structured Information Sources. In *Proc. ACM SIGMOD Workshop on Management of Semi-structured Data*, 1997.

[Bartlett & Williamson 91] Bartlett, P. and Williamson, R. Investigating the distributional assumptions of the PAC learning model. In *Proc. 4th Workshop Computational Learning Theory*, pages 24–32, 1991.

[Benedek & Itai 88] Benedek, G. and Itai, A. Learnability by fixed distributions. In *Proc. 1st Workshop Computational Learning Theory*, pages 80–90, 1988.

[Biermann & Feldman 72] Biermann, A. and Feldman, J. On the synthesis of finite state machines from samples of their behavior. *IEEE Trans. on Computers*, C-21:592–7, 1972.

[Blumer et al. 89] Blumer, A., Ehrenfeucht, A., Haussler, D., and Warmuth, M. Learnability and the Vapnik-Chervónenkis dimension. *J. ACM*, 36(4):929–65, 1989.

[Borgman & Siegfried 92] Borgman, C. and Siegfried, S. Getty's Synoname and its cousin: A survey of applications of personal name-matching algorithms. *J. Amer. Soc. of Information Science*, 43:459–76, 1992.

[Bowman et al. 94] Bowman, M., Danzig, P., Manber, U., and Schwartz, F. Scalable Internet discovery: Research problems and approaches. *C. ACM*, 37(8):98–107, 1994.

[Bradshaw 97] Bradshaw, J., editor. *Intelligent Agents.* MIT Press, 1997.

[Brodie & Stonebraker 95] Brodie, M. and Stonebraker, M. *Migrating Legacy Systems: Gateways, Interfaces, & the Incremental Approach.* Morgan Kaufmann, 1995.

[Carey et al. 95] Carey, M., Haas, L., Schwarz, P., Arya, M., Cody, W., Fagin, R., Flickner, M., Luniewski, A., Niblack, W., Petkovic, D., Thomas, J., Williams, J., and Wimmers, E. Towards heterogeneous multimedia information systems: The Garlic approach. In *Proc. 5th Int. Workshop of Research Issues in Data Engineering: Distributed Object Management*, pages 124–31, 1995.

[Chawathe et al. 94] Chawathe, S., Garcia-Molina, H., Hammer, J., Ireland, K., Papakonstantinou, Y., Ullman, J., and Widom, J. The TSIMMIS project: Integration of heterogeneous information sources. In *Proc. 10th Meeting of the Information Processing Soc. of Japan*, pages 7–18, 1994.

[Chidlovskii et al. 97] Chidlovskii, B., Borghoff, U., and Chevalier, P. Towards Sophisticated Wrapping of Web-based Information Repositories. In *Proc. Conf. Computer-Assisted Information Retrieval*, pages 123–35, 1997.

[Cohen & Singer 96] Cohen, W. and Singer, W. Learning to Query the Web. In *Proc. Workshop Internet-based Information Systems, 13th Nat. Conf. Artificial Intelligence*, pages 16–25, 1996.

[Collet et al. 91] Collet, C., Huhns, M., and Shen, W. Resource integration using a large knowledge base in CARNOT. *IEEE Computer*, 1991.

[Cowie & Lehnert 96] Cowie, J. and Lehnert, W. Information extraction. *C. ACM*, 39(1):80–91, 1996.

[Decatur & Gennaro 95] Decatur, S. and Gennaro, R. On Learning from Noisy and Incomplete Examples. In *Proc. 8th Annual ACM Conf. Computational Learning Theory*, pages 353–60, 1995.

[Decker et al. 97] Decker, K., Sycara, K., and Williamson, M. Middle-agents for the internet. In *Proc. 15th Int. Joint Conf. AI*, pages 578–83, 1997.

[Dietterich & Michalski 83] Dietterich, T. and Michalski, R. A comparative review of selected methods for learning from examples. In [Michalski et al. 83], chapter 3, pages 41–82.

[Doorenbos 97] Doorenbos, R., October 1997. Personal communication.

[Doorenbos et al. 97] Doorenbos, R., Etzioni, O., and Weld, D. A scalable comparison-shopping agent for the World-Wide Web. In *Proc. Autonomous Agents*, pages 39–48, 1997.

[Douglas & Hurst 96] Douglas, S. and Hurst, M. Layout and language: Lists and tables in tehcnical documents. In *Proc. SIGPARSE*, pages 19–24, 1996.

[Douglas et al. 95] Douglas, S., Hurst, M., and Quinn, D. Using Natural Language Processing for Identifying and Intepreting Tables in Plain Text. In *Proc. 4th Symp. Document Analysis and Information Retrieval*, pages 535–46, 1995.

[Draper 97] Draper, D., October 1997. Personal communication.

[Etzioni & Weld 94] Etzioni, O. and Weld, D. A softbot-based interface to the Internet. *C. ACM*, 37(7):72–6, 1994.

[Etzioni 93] Etzioni, O. Intelligence without robots: A reply to Brooks. *AI Magazine*, 14(4):7–13, 1993.

[Etzioni 96a] Etzioni, O. Moving up the information food chain: softbots as information carnivores. In *Proc. 13th Nat. Conf. AI*, 1996.

[Etzioni 96b] Etzioni, O. The World Wide Web: quagmire or gold mine? *C. ACM*, 37(7):65–8, 1996.

[Etzioni et al. 93] Etzioni, O., Lesh, N., and Segal, R. Building softbots for UNIX (preliminary report). Technical Report 93-09-01, University of Washington, 1993.

[Etzioni et al. 94] Etzioni, O., Maes, P., Mitchell, T., and Shoham, Y., editors. *Working Notes of the AAAI Spring Symposium on Software Agents*, Menlo Park, CA, 1994. AAAI Press.

[Finin et al. 94] Finin, T., Fritzson, R., McKay, D., and McEntire, R. KQML: A language and protocol for knowledge and information exchange. In *Knowledge Building and Knowledge Sharing*. Ohmsha and IOS Press, 1994.

[Florescu et al. 95] Florescu, D., Rashid, L., and Valduriez, P. Using heterogeneous equivalences for query rewriting in multi-database systems. In *Proc. Cooperative Information Systems*, 1995.

[Freitag 96] Freitag, D. Machine Learning for Information Extraction from Online Documents: A Preliminary Experiment. Unpublished manuscript, 1996. Available at www.cs.cmu.edu/afs/cs.cmu.edu/user/dayne/www/prelim.ps.

[Freitag 97] Freitag, D. Using grammatical inference to improve precision in information extraction. In *Working Papers of the Workshop on Automata Induction, Grammatical Inference, and Language Acquisition, 14th Int. Conf. Machine Learning*, 1997. Available at http://www.cs.cmu.edu/˜pdupont/-ml97p/ml97_GI_wkshp.tar.

[Friedman & Weld 97] Friedman, M. and Weld, D. Efficiently executing information-gathering plans. In *Proc. 15th Int. Joint Conf. AI*, pages 785–91, 1997.

[Goan et al. 96] Goan, T., Benson, N., and Etzioni, O. A grammar inference algorithm for the world wide web. In *Proc. AAAI Spring Symposium on Machine Learning in Information Access*, 1996.

[Gold 78] Gold, E. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978.

[Green & Krishnamoorthy 95] Green, E. and Krishnamoorthy, M. Model-Based Analysis of Printed Tables. In *Proc. 3rd Int. Conf. Document Analysis and Recognition*, 1995.

[Gupta 89] Gupta, A., editor. *Integration of Information Systems: Bridging Heterogeneous Databases.* IEEE Press, 1989.

[Haussler 88] Haussler, D. Quantifying inductive bias. *J. Artificial Intelligence*, 36(2):177–221, 1988.

[Hayes 94] Hayes, P. NameFinder: Softwre that finds names in text. In *Proc. Conf. Computer-Assisted Information Retrieval*, pages 762–74, 1994.

[Hobbs 92] Hobbs, J. The generic information extraction system. In *Proc. 4th Message Understanding Conf.*, 1992.

[Kearns & Vazirani 94] Kearns, M. and Vazirani, U. *An introduction to computational learning theory.* MIT, 1994.

[Kearns 93] Kearns, M. Efficient Noise-Tolerant Learning from Statistical Queries. In *Proc. 25th Annual ACM Symp. Theory of Computing*, pages 392–401, 1993.

[Kirk et al. 95] Kirk, T., Levy, A., Sagiv, Y., and Srivastava, D. The Information Manifold. In *AAAI Spring Symposium: Information Gathering from Heterogeneous, Distributed Environments*, pages 85–91, 1995.

[Knuth et al. 77] Knuth, D., Morris, J., and Pratt, V. Fast pattern matching in strings. *SIAM J. Computing*, 6(2):323–50, 1977.

[Krulwich 96] Krulwich, B. The BargainFinder agent: Comparison price shopping on the Internet. In Williams, J., editor, *Bots and Other Internet Beasties*, chapter 13. SAMS.NET, 1996.

[Kwok & Weld 96] Kwok, C. and Weld, D. Planning to gather information. In *Proc. 13th Nat. Conf. AI*, 1996.

[Levy et al. 96] Levy, A., Rajaraman, A., and Ordille, J. Query-answering algorithms for information agents. In *Proc. 13th Nat. Conf. AI*, 1996.

[Luke et al. 97] Luke, S., Spector, L., Rager, D., and Hendler, J. Ontology-based web agents. In *Proc. First Int. Conf. Autonomous Agents*, 1997.

[Martin & Biggs 92] Martin, A. and Biggs, N. *Computational learning theory: An introduction.* Cambridge University Press, 1992.

[Michalski 83] Michalski, R. A theory and methodology of inductive learning. In [Michalski et al. 83], chapter 4, pages 83–135.

[Michalski et al. 83] Michalski, R., Carbonell, J., and Mitchell, T., editors. *Machine Learning — An Artificial Intelligence Approach.* Morgan Kaufman, 1983.

[Mitchell 80] Mitchell, T. The need for biases in learning generalizations. Technical Report CBM-TR-117, Dept. of Computer Science, Rutgers Univ., 1980.

[Mitchell 97] Mitchell, T. *Machine Learning.* McGraw Hill, 1997.

[Monge & Elkan 96] Monge, A. and Elkan, C. The field matching problem: Algorithms and applications. In *Proc. 2nd Int. Conf. Knowledge Discovery and Data Mining*, 1996.

[Paik et al. 93] Paik, W., Liddy, E., Yu, E., and McKenna, M. Categorizing and standardizing proper nouns for efficient retrieval. In *Proc. Assoc. for Computational Linguistics Workshop on the Aquisition of Lexical Knowledge from Text*, 1993.

[Papakonstantinou et al. 95] Papakonstantinou, Y., Garcia-Monlina, H., and Widom, J. Object exchange across heterogeneous information sources. In *Proc. 11th Int. Conf. Data Engineering*, pages 251–60, 1995.

[Perkowitz & Etzioni 95] Perkowitz, M. and Etzioni, O. Category translation: Learning to understand information on the Internet. In *Proc. 14th Int. Joint Conf. AI*, pages 930–6, 1995.

[Perkowitz et al. 97] Perkowitz, M., Doorenbos, R., Etzioni, O., and Weld, D. Learning to understand information on the Internet: An example-based approach. *J. Intelligent Information Systems*, 8(2):133–153, 1997.

[Rau 91] Rau, L. Extracting company names from text. In *Proc. 9th Nat. Conf. AI*, 1991.

[Riloff 93] Riloff, E. Automatically Constructing a Dictionary for Information Extraction Tasks. In *Proc. 11th Nat. Conf. AI*, pages 811–6, 1993.

[Roth & Schwartz 97] Roth, M. and Schwartz, P. Don't scrap it, wrap it! A wrapper architecture for legacy data sources. In *Proc. 22nd VLDB Conf.*, pages 266–75, 1997.

[Rus & Subramanian 97] Rus, D. and Subramanian, D. Customizing information capture and access. *ACM Trans. Information Systems*, 15(1):67–101, 1997.

[Russell & Norvig 95] Russell, S. and Norvig, P. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 1995.

[Schuurmans & Greiner 95] Schuurmans, D. and Greiner, R. Practical PAC Learning. In *Proc. 14th Int. Joint Conf. AI*, pages 1169–75, 1995.

[Selberg & Etzioni 95] Selberg, E. and Etzioni, O. Multi-service search and comparison using the metacrawler. In *Proc. 4th World Wide Web Conf.*, pages 195–208, Boston, MA USA, 1995.

[Selberg & Etzioni 97] Selberg, E. and Etzioni, O. The metacrawler architecture for resource aggregation on the web. *IEEE Expert*, 12(1):8–14, January 1997.

[Shakes et al. 97] Shakes, J., Langheinrich, M., and Etzioni, O. Dynamic reference sifting: a case study in the homepage domain. In *Proc. 6th World Wide Web Conf.*, 1997. See `http://www.cs.washington.edu/research/ahoy`.

[Shklar et al. 94] Shklar, L., Thatte, S., Marcus, H., and Sheth, A. The InfoHarness Information Integration Platform. In *Proc. 2nd Int. WWW Conf.*, 1994.

[Shklar et al. 95] Shklar, L., Shah, K., and Basu, C. Putting Legacy Data on the Web: A Repository Definition Language. In *Proc. 3rd Int. WWW Conf.*, 1995.

[Smeaton & Crimmins 97] Smeaton, A. and Crimmins, F. Relevance Feedback and Query Expansion for Searching the Web: A Model for Searching a Digital Library. In *Proc. 1st European Conf. Digital Libraries*, pages 99–112, 1997.

[Soderland 97a] Soderland, S., October 1997. Personal communication.

[Soderland 97b] Soderland, S. *Learning Text Analysis Rules for Domain-Specific Natural Language Processing.* PhD dissertation, University of Massachusetts, 1997.

[Soderland 97c] Soderland, S. Learning to Extract Text-based Information from the World Web. In *Proc. 3rd Int. Conf. Knowledge Discovery and Data Mining*, 1997.

[Soderland et al. 95] Soderland, S., Fisher, D., Aseltine, J., and Lehnert, W. CRYSTAL: Inducing a Conceptual Dictionary. In *Proc. 14th Int. Joint Conf. AI*, pages 1314–21, 1995.

[Tanida & Yokomori 92] Tanida, N. and Yokomori, T. Polynomial-time identification of strictly regular languages in the limit. *IEICE Tran. Information and Systems*, E75-D(1):125–32, 1992.

[Tejada et al. 96] Tejada, S., Knoblock, C., and Minton, S. Learning models for multi-source integration. In *Proc. AAAI Spring Symposium on Machine Learning in Information Access*, 1996.

[Valiant 84] Valiant, L. A theory of the learnable. *C. ACM*, 27(11):1134–42, 1984.

[Vapnik & Chervónenkis 71] Vapnik, V. and Chervónenkis, A. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications*, 16(2):264–80, 1971.

[Wooldridge & Jennings 95] Wooldridge, M. and Jennings, N., editors. *Intelligent Agents.* Springer Verlag, 1995.

[Zaiane & Jiawei 95] Zaiane, O. R. and Jiawei, H. Resource and knowledge discovery in global information systems: A preliminary design and experiment. In *Proc. KDD*, pages 331–6, 1995.

VITA

Nicholas Kushmerick received his B.S. degree in Computer Engineering, with University Honors, from Carnegie Mellon University in 1989. From 1989–91, he worked as a research programmer in CMU's Psychology Department. From 1991–97, he was a graduate student in the University of Washington's Department of Computer Science and Engineering. He received his M.S. degree in 1994 and his Ph.D. degree in 1997. In 1994–95, he was a visiting researcher at the Center for Theoretical Study (Charles University) and the Institute for Information Theory and Automation (Czech Academy of Science) in Prague. In 1998, Nick will join the Department of Computer Applications at Dublin City University as a Lecturer.