BUILDING INTELLIGENT AGENTS THAT LEARN TO RETRIEVE AND EXTRACT INFORMATION

By

Tina Eliassi-Rad

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

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN – MADISON

2001

For Branden

Abstract

The rapid growth of on-line information has created a surge of interest in tools that are able to retrieve and extract information from on-line documents. In this thesis, I present and evaluate a computer system that rapidly and easily builds instructable and self-adaptive software agents for both the information retrieval (IR) and the information extraction (IE) tasks.

My system is called WAWA (short for *Wisconsin Adaptive Web Assistant*). WAWA interacts with the user and an on-line (textual) environment (*e.g.*, the Web) to build an intelligent agent for retrieving and extracting information. WAWA has two sub-systems: (*i*) an information retrieval (IR) sub-system, called *WAWA-IR*; and, (*ii*) an information extraction (IE) sub-system, called *WAWA-IE*. WAWA-IR is a general search-engine agent, which can be trained to produce specialized and personalized IR agents. WAWA-IE is a general extractor system, which creates specialized agents that accurately extract pieces of information from documents in the domain of interest.

WAWA utilizes a *theory-refinement* approach to build its intelligent agents. There are four four primary advantages of using such an approach. First, WAWA's agents are able to perform reasonably well initially because they are able to utilize users' prior knowledge. Second, users' prior knowledge does not have to be correct since it is refined through learning. Third, the use of prior knowledge, plus the continual dialog between the user and an agent, decreases the need for a large number of training examples because training is not limited to a binary representation of positive and negative examples. Finally, WAWA provides an appealing middle ground between non-adaptive agent programming languages and systems that solely learn user preferences from training examples.

WAWA's agents have performed quite well in empirical studies. WAWA-IR experiments demonstrate the efficacy of incorporating the feedback provided

by the Web into the agent's neural networks to improve the evaluation of potential hyperlinks to traverse. Wawa-IE experiments produce results that are competitive with other state-of-art systems. Moreover, they demonstrate that Wawa-IE agents are able to intelligently and efficiently select from the space of possible extractions and solve *multi-slot* extraction problems.

Acknowledgements

I am grateful to the following people and organizations:

- My advisor, Jude Shavlik, for his constant support, guidance, and generosity throughout this research.
- My committee members, Mark Craven, Olvi Mangasarian, David Page, and Louise Robbins, for their time and consideration.
- My husband, Branden Fitelson, for his unwavering love and support during the past 8.5 years.
- My parents for their love, support, generosity, and belief in my abilities.
- My brothers, Babak and Nima, for being wonderful siblings.
- Zari and the entire Hafez family for their constant kindness over the past 13 years.
- Anne Condon for being a great mentor since spring of 1992.
- Peter Andreae for his useful comments on this research.
- Rich Maclin for being supportive whenever our paths crossed.
- Lorene Webber for all her administrative help.
- National Science Foundation (NSF) grant IRI-9502990, National Library of Medicine (NLM) grant 1 R01 LM07050-01, and University of Wisconsin Vilas Trust for supporting this research.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

The exponential growth of on-line information (Lawrence and Giles 1999) has increased demand for tools that are able to efficiently retrieve and extract *textual* information from on-line documents. In a ideal world, you would be able to instantaneously retrieve precisely the information you want (whether it is a whole document or fragments of it). What is the next best option? Consider having an assistant, which rapidly and easily builds instructable and self-adaptive software agents for both the information retrieval (IR) and the information extraction (IE) tasks. These intelligent software agents would learn your interests and automatically refine their models of your preferences over time. Their mission would be to spend 24 hours a day looking for documents of interest to you and answering specific questions that you might have. In this thesis, I present and evaluate such an assistant.

1.1 Wisconsin Adaptive Web Assistant

My assistant is called WAWA (short for *Wisconsin Adaptive Web Assistant*). WAWA interacts with the user and an on-line (textual) environment (*e.g.*, the Web) to build an intelligent agent for retrieving and/or extracting information. Figure 1 illustrates an overview of WAWA. WAWA has two sub-systems: (*i*) an information-retrieval sub-system, called *Wawa-IR*; and, (*ii*) an information-extraction sub-system, called *WAWA-IE*. WAWA-IR is a general search-engine agent, which can be trained to produce specialized and personalized IR agents.

WAWA-IE is a general extractor system, which creates specialized agents that extract pieces of information from documents in the domain of interest.

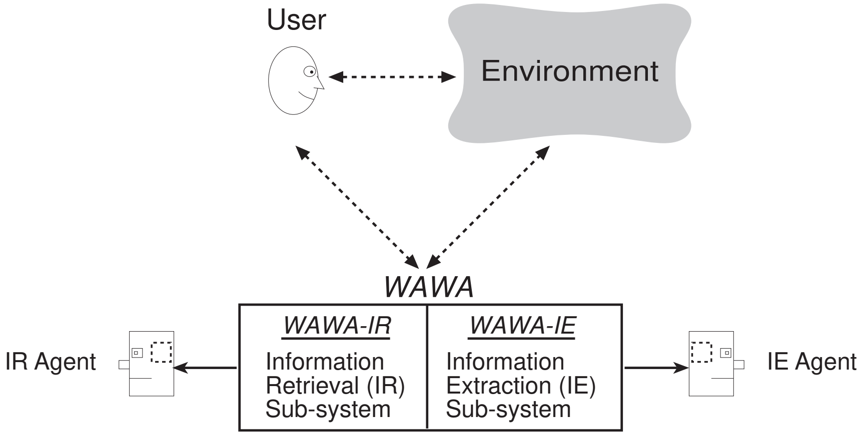

Figure 1: An Overview of WAWA

WAWA builds its agents based on ideas from the theory-refinement community within machine learning (Pazzani and Kibler 1992; Ourston and Mooney 1994; Towell and Shavlik 1994). Users specify their prior knowledge about the desired task. This knowledge is then "compiled" into "knowledge based" neural networks (Towell and Shavlik 1994), thereby allowing subsequent refinement whenever training examples are available. The advantages of using a theory-refinement approach to build intelligent agents are as follows:

- WAWA's agents are able to perform reasonably well initially because they are able to utilize users' prior knowledge.

- Users' prior knowledge does not have to be correct since it is refined through learning.

- The use of prior knowledge, plus the continual dialog between the user and an agent, decreases the need for a large number of training examples because human-machine communication is not limited to solely providing positive and negative examples.

- WAWA provides an appealing middle ground between non-adaptive agent programming languages (Etzioni and Weld 1995; Wooldridge and Jennings 1995) and systems that solely learn user preferences from training examples (Pazzani, Muramatsu, and Billsus 1996; Joachims, Freitag, and Mitchell 1997).

WAWA's agents are arguably intelligent because they can adapt their behavior according to the users' instructions and the feedback they get from their environments. Specifically, they are learning agents that use neural networks to store and modify their knowledge. Figure 2 illustrates the interaction between the user, an intelligent (WAWA) agent, and the agent's environment. The user[1] observes the agent's behavior (*e.g.*, the quality of the pages retrieved) and provides helpful instructions to the agent. Following Maclin and Shavlik (1996), I refer to users' instructions as *advice*, since this name emphasizes that the agent does not blindly follow the user-provided instructions, but instead refines the advice based on its experiences. The user inputs his/her advice into a user-friendly *advice interface*. The given advice is then processed and mapped into the agent's knowledge base (*i.e.*, its two neural networks), where it gets refined based on the agent's experiences. Hence, the agent is able to represent the user model in its neural networks, which have representations for which effective learning algorithms are known (Mitchell 1997).

[1]I envision that there are two types of potential users of WAWA: (1) *application developers*, who build an intelligent agent on top of WAWA and (2) *application users*, who use the resulting agent. (When I use the phrase *user* in this thesis, I mean the former.) Both types of users can provide advice to the underlying neural networks, but I envision that usually the application users will indirectly do this through some specialized interface that the application developers create. A scenario like this is discussed in Chapter 5.

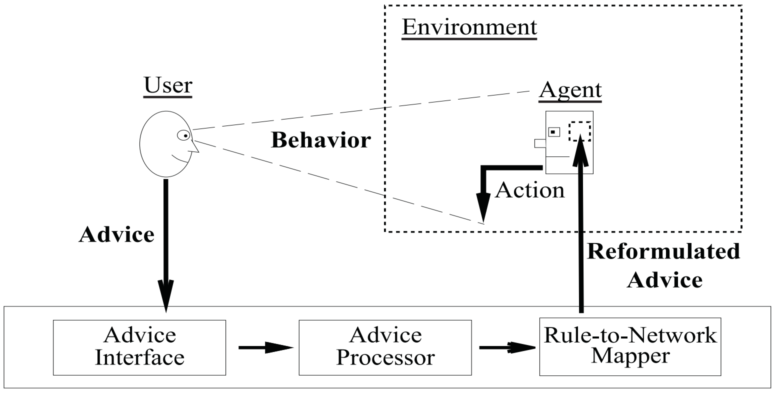

Figure 2: The Interaction between a User, an Intelligent Agent, and the Agent's Environment

1.2 Thesis Statement

In this thesis, I present and evaluate WAWA, which is a system for rapidly building intelligent software agents that retrieve *and* extract information. The thesis of this dissertation is as follows:

> *Theory-refinement techniques are quite useful in solving information-retrieval and information-extraction tasks. Agents using such techniques are able to (i) produce reasonable performance initially (without requiring that the knowledge provided by the user be 100% correct) and (ii) reduce the burden on the user to provide training examples (which are tedious to obtain in both tasks).*

1.3 Thesis Overview

This thesis is organized as follows. Chapter 2 provides the necessary background for the concepts and techniques used in WAWA. I present WAWA's fundamental operations in Chapter 3. WAWA's information-retrieval (WAWA-IR) sub-system and its case studies are discussed in Chapters 4 and 5, respectively.

Wawa's information-extraction (Wawa-IE) sub-system along with its experimental studies are discussed in Chapters 6 and 7, respectively. Related work is presented in Chapter 8. I discuss the contributions of this thesis, Wawa's limitations, some future directions, and concluding remarks in Chapter 9. Appendix A presents Wawa's advice language in its entirety. Appendices B, C, D, and E provide the advice rules used in the empirical studies done on Wawa-IR and Wawa-IE.

Chapter 2

Background

This chapter provides the necessary background for the concepts used in WAWA. These concepts are multi-layer feed-forward neural networks, knowledge-based neural networks, reinforcement learning, information retrieval, and information extraction. Readers who are familiar with these topics may wish to skip this chapter.

2.1 Multi-Layer Feed-Forward Neural Networks

Multi-layer feed-forward neural networks learn to recognize patterns. Figure 3 shows a two-layer feed-forward network.[1]

Given a set of input vectors and their corresponding output vectors, (X,Y), a multi-layer feed forward neural network can be trained to learn a function, f, which maps new input vectors, X', into the corresponding output vectors, Y'.

The ability to capture nonlinear functions makes multi-layer feed-forward neural networks powerful. Activation functions are used on the hidden units of a feed-forward neural network to introduce nonlinearity into the network. There are three commonly used activation functions: (i) the step function, (ii) the sign function, and (iii) the sigmoid function. The step function returns 1 if the weighted sum of its inputs is greater than or equal to some pre-defined threshold, t; otherwise, it returns 0. The sign function returns 1 if the weighted

[1] Figure 3 was adapted from Rich and Knight (1991), page 502.

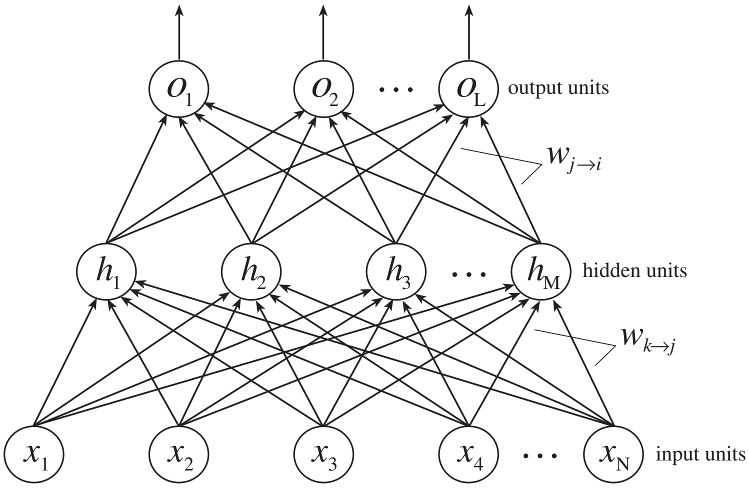

Figure 3: A Two-Layer Feed-Forward Network

sum of its inputs is greater than or equal to zero; otherwise, it returns 0. The sigmoid function returns $\frac{1}{1+e^{-in_i}}$ where in_i is the weighted sum of inputs into unit i plus the pre-defined bias on unit i. Activation functions are also used on the output units to capture the distribution of the output values.

The most popular method of training multi-layer feed-forward networks is called backpropagation (BP) (Rumelhart, Hinton, and Williams 1986). Table 1 describes the BP algorithm. The sigmoid function is a popular activation function for backpropagation learning since it is differentiable.

A good stopping criteria for BP training is to set aside some of the examples in the training set into a new set (known as the tuning set). Whenever the accuracy on the tuning set starts decreasing, we are overfitting the training data and should stop training.

In WAWA, the output units output the weighted sum of their inputs and the activation function for the hidden units is sigmoidal.

Table 1: The Backpropagation Algorithm

Inputs: a multi-layer feed-forward network, a set of input/output pairs (known as the training set), and a learning rate (η)

Output: a trained multi-layer feed-forward network

Algorithm:

- initialize all weights to small random numbers
- repeat until stopping criteria has been met
  - for each example $\langle X, Y \rangle$ in the training set do the following
    - compute the output, O, for this example by instantiating X into the input units and doing forward-propagation
    - compute the error at the output units, $E = Y - O$
    - update the weights into the output units,
      $W_{j \to i} = W_{j \to i} + \eta \times a_j \times E_i \times g'(in_i)$
      where $a_j = g(in_j)$ is the activation of unit j,
      $g'(in_i)$ is the derivative of the activation function g, and
      $in_i = (\sum_j W_{j \to i} \times a_j) + bias_i$
    - for each hidden layer in the network do the following
      - compute the error at each node,
        $\Delta_j = g'(in_j) \sum_i W_{j \to i} \times \Delta_i$ where $\Delta_i = E_i \times g'(in_i)$
      - update the weights into the hidden layer,
        $W_{k \to j} = W_{k \to j} + \eta \times X_k \times \Delta_j$
        where X_k is the activation of the j^{th} unit in the input vector

2.2 Knowledge-Based Artificial Neural Networks

A knowledge-based artificial neural network (KBANN) allows a user to input prior knowledge into a multi-layer feed-forward neural network (Towell and Shavlik 1994). The user expresses her prior knowledge by a set of propositional rules. Then, an AND-OR dependency graph is constructed. Each node in the AND-OR dependency graph becomes a unit in KBANN. Additional units are added for OR nodes. The biases of each AND unit and the weights coming into the AND unit are set such that the unit will get activated only when all of its inputs are true. Similarly, the biases of each OR unit and the weights coming into the OR unit are initialized such that the unit will get activated only when at least one of its inputs is true. Links with low weights are added between layers of the network to allow learning over the long run.

Figure 4 illustrates the KBANN algorithm.[2] In part (a), some prior knowledge is given in the form of propositional rules. Part (b) shows the AND-OR dependency graph for the rules given in part (a). In part (c), each node in the graph becomes a unit in the network and appropriate weights are added to the links to capture the semantics of each rule. For example, the weights on links $X \rightarrow Z$ and $Y \rightarrow Z$ are set to 5 and the bias[3] of the unit Z is set to -6. This means that unit Z will be true if and only if both units X and Y are true. To enable future learning, low-weighted links are added to the network in part (d).

After KBANN is constructed, we can apply the backpropagation algorithm (see Section 2.1) to learn from the training set.

---

[2]Figure 4 was adapted from Maclin (1995), page 17.

[3]For an `AND` unit, the bias equals $5(\#unnegated\_antecedents - 0.5)$. For an `OR` unit, the bias equals $-5(\#negated\_antecedents - 0.5)$. KBANN uses sigmoidal activation function on its units. If the activation function outputs a value ≥ 0.5, then the unit is activated. Otherwise, the unit is not activated.

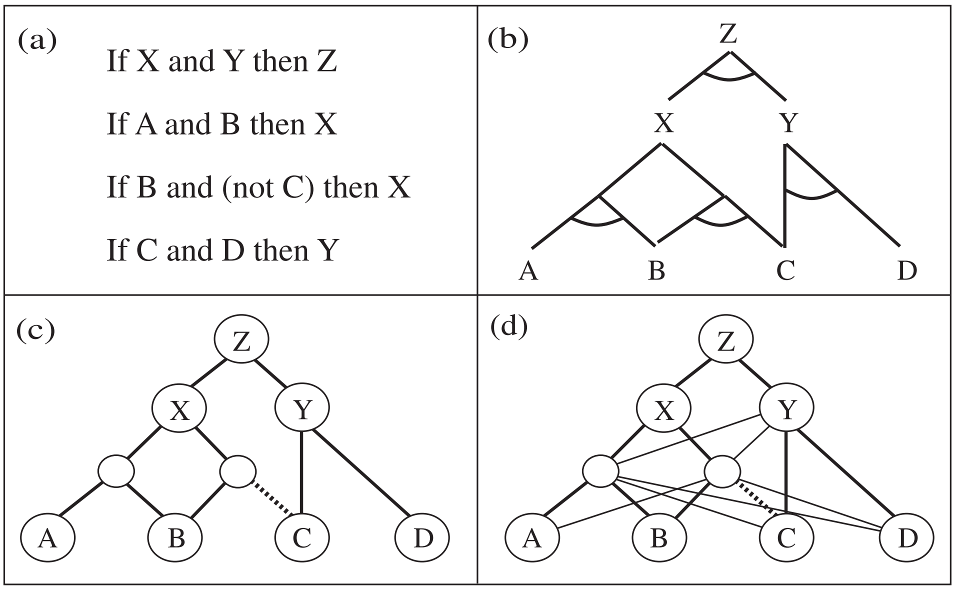

Figure 4: An Example of the KBANN Algorithm. Box (a) contains a set of propositional rules. Box (b) shows an AND-OR dependency graph for rules in box (a). In box (c), each node in the AND-OR graph is mapped to a unit. The weights on links and the biases on units are set such that they reflect either an AND gate or an OR gate. Additional units are added to capture the functionality of an OR gate. In box (d), links with low weights are added between layers to allow future learning.

2.3 Reinforcement Learning

In a reinforcement-learning (RL) paradigm, the learner attempts to learn a course of actions which maximizes the total rewards he/she receives from the environment. A RL system has five major elements (Sutton and Barto 1998):[4]

- An *agent* which is the learner in the RL paradigm.
- An *environment* which is a collection of states. The agent is able to

[4]Some RL systems have an optional element which describes the behavior of the environment. This element is called the *model of the environment*. Given a state and an action, it predicts the next state and the immediate reward. WAWA does not use a model of the environment, since it is a daunting task to model the World-Wide Web.

interact with the environment by performing actions which take the agent from one state to the other.

- A *policy* which specifies a mapping from the agent's current state to its chosen action (*i.e.*, the agent's behavior at a given time).
- A *reward function* which returns the immediate reward the agent receives for performing a particular action at a particular state in the environment.
- A *value function* which defines the predicted goodness of an action in the long run.

WAWA uses a class of RL methods called *temporal-difference* learning (Sutton 1988). In particular, WAWA uses a method called *Q-learning* (Watkins 1989). In Q-learning, a system tries to learn a function called the *Q-function.* This function takes as input a state, s, and an action, a and outputs the sum of the immediate reward the agent will receive for performing action a from state s and the discounted value of the optimal policy taken from the resultant state, s'. The optimal policy from s' is the action, a', which maximizes the Q-function with inputs s' and a'. Therefore, we can write the Q-function as:

$$Q(s,a) = r(s,a) + \gamma * \underset{a'}{max} Q(s',a')$$

where $r(s,a)$ is the immediate reward for taking action a from state s, and γ is the discounted value for rewards in the future.

WAWA's information retrieval sub-system learns a modified version of the Q-function by using a knowledge-based neural network called *ScoreLink* (see Section 4.3 for details). ScoreLink learns to predict which links to follow. The output of ScoreLink is $\hat{Q}(s,a)$, which is WAWA's estimate of the actual $Q(s,a)$. ScoreLink learns to predict the value of a link, l, by backpropagating on the difference between the value of l before and after fetching the page, p, to which it points.

2.4 Information Retrieval

Information retrieval (IR) systems take as input a set of documents (*a.k.a.*, the corpus) and a query (*i.e.*, a set of keywords). They return documents from the corpus that are *relevant* to the given query.

Most IR systems preprocess documents by removing commonly used words (*a.k.a.*, "stop" words) and stemming words (*e.g.*, replacing "walked" with "walk"). After the preprocessing phase, word-order information is lost and the remaining words are called *terms*. Then, a term × document matrix is created, where the documents are the rows of the matrix and the terms are the columns of the matrix. Table 2 depicts such a matrix.[5]

Table 2: A Term × Document Matrix

	$term_1$	$term_2$	$\dots$	$term_j$	$\dots$	$term_m$
doc_1	w_{11}	w_{21}	$\dots$	w_{1j}	$\dots$	w_{1m}
doc_2	w_{21}	w_{22}	$\dots$	w_{2j}	$\dots$	w_{2m}
$\vdots$	$\vdots$	$\vdots$	$\dots$	$\vdots$	$\dots$	$\vdots$
doc_i	w_{i1}	w_{i2}	$\dots$	w_{ij}	$\dots$	w_{im}
$\vdots$	$\vdots$	$\vdots$	$\dots$	$\vdots$	$\dots$	$\vdots$
doc_n	w_{n1}	w_{n2}	$\dots$	w_{nj}	$\dots$	w_{nm}

The entry w_{ij} is commonly defined as follows:

$$w_{ij} = TF(term_j, doc_i) \times \log(\tfrac{|D|}{DF(term_j)})$$

where the function $TF(term_j, doc_i)$ returns the number of times $term_j$ appears in document doc_i, $|D|$ is the number of documents in the corpus, and the function $DF(term_j)$ returns the number of times $term_j$ appears in all the documents in the corpus. This representation of documents is known as the *vector model* or the *bag-of-words* representation (Salton 1991) and the weighing strategy is known as TF/IDF, short for term frequency/inverse document frequency (Salton and Buckley 1988).

[5]Table 2 was adapted from Rose (1994), page 66.

In the vector model, users' queries are also represented as term vectors. IR systems using the vector model employ similarity measures to find relevant documents to a query. A common similarity measure is the *cosine measure*, where the relevance of a document to the user's query is measured by the cosine of the angle between the document vector and the query vector. The smaller the cosine, the more relevant the document is considered to be to the user's query.

IR systems are commonly evaluated by two measures: *precision* and *recall*. Recall represents the percentage of relevant documents that are retrieved from the corpus. Precision represents the percentage of relevant documents that are in the retrieved documents. Their definitions are:

$$Precision = \frac{|RELEVANT \cap RETRIEVED|}{|RETRIEVED|}$$

$$Recall = \frac{|RELEVANT \cap RETRIEVED|}{|RELEVANT|}$$

where *RELEVANT* represents the set of documents in our corpus that are relevant to a particular query and *RETRIEVED* is the set of documents retrieved for that query.

An IR system tries to maximizes both recall and precision. The F_β-*measure* combines precision and recall and is defined to be $(\frac{(\beta^2+1.0)\cdot Recall\cdot Precision}{(\beta^2\cdot Precision)+Recall})$, where $\beta \in [0,1]$. When $\beta = 1$, precision and recall are given the same weight in the F_β-measure. The F_1-measure is more versatile than either precision or recall for explaining relative performance of different systems, since it takes into account the inherent tradeoff that exists between precision and recall.

Most IR systems categorize a document as either relevant or irrelevant. Instead of taking this black or white view of the world, WAWA learns to rate Web documents on a scale of -10.0 to 10.0. This numeric scale is then mapped into five categories (*perfect*, *good*, *okay*, *indifferent*, and *terrible*) in order to simplify the task of labeling (Web) pages for users. Table 3 illustrates this mapping.

Table 3: Mapping between Categories Assigned to Web Pages and Their Numeric Scores

Categories for Web Pages	**Scores of Web Pages**
Perfect	$(\ \ 9.0,\ 10.0]$
Good	$(\ \ 5.0,\ \ 9.0]$
Okay	$(\ \ 0.0,\ \ 5.0]$
Indifferent	$(-3.0,\ \ 0.0]$
Terrible	$[-10.0, -3.0]$

2.5 Information Extraction

Information extraction (IE) systems are typically given a set of documents and a set of slots in a pre-defined template. They are supposed to extract phrases that accurately fill the slots in the given template (such phrases are refered to as "slot fillers"). To accomplish this task, some IE systems attempt to learn the extraction patterns from training documents. These patterns (*a.k.a.*, rules) are then applied to future documents to extract some assertions about those documents. For example, the following pattern extracts dates like "Sunday, August 1, 1999", "Sun, Aug 1, 99", and "Sunday, Aug 1, 1999", etc:

"*Day [ ]* , [ ]* Month [ ]* DayNum [ ]* , [ ]* Year*"

where *[ ]** = 0 or more spaces, *Day* = [Monday | ... | Sunday | Mon | ... | Sun], *Month* = [January | ... | December | Jan | ... | Dec], *DayNum* = [1 | 2 | ... | 30 | 31], *Year* = [*Digit Digit Digit Digit* | *Digit Digit*], and *Digit* = [0 | 1 | ... | 9]. The notation *N*=[A | B | $\cdots$] lists the possible values for variable *N*. For example, *SEX* = [Male | Female].

The documents given to an IE system can have different text styles (Soderland 1999). They can be either structured (*e.g.*, pages returned by a yellow-pages telephone directory), semi-structured (*e.g.*, seminar announcements), or free (*e.g.*, news articles).

There are two types of IE systems. The first type is *individual-slot* (or *single-slot*) systems, which produce a single filled template for each document. The second type is *combination-slots* (or *multi-slot*) systems, which produce more than one filled template for each document. The multi-slot extraction problem is harder than the single-slot extraction problem since the slot fillers in the template depend on each other. For example, suppose the IE task is to extract proteins and their locations in the cell from a set of biological documents, where each document contains multiple instances of proteins and their locations in the cell. In this problem, the IE system must be able to match each protein with its location in the cell. Giving the user a list of proteins and a separate list of locations is useless.

The input requirements and the syntactic structure of the learned patterns vary substantially from one IE system to the next. See Section 8.3 for a discussion of different IE systems.

Chapter 3

WAWA's Core

This chapter presents Wawa's fundamental operations,[1] which are used in both the IR and the IE sub-systems of Wawa (see Chapters 4 and 6 for details on the IR and the IE sub-systems, respectively). These operations include extracting features from Web pages,[2] handling of Wawa's advice language, and scoring arbitrarily long pages with neural networks. Figure 5 illustrates how an agent uses these operations to score a page. The *page processor* gets a page from the environment (*e.g.*, the Web) and produces an internal representation of the page (by extracting features from it). This new representation of the page is then given to the agent's knowledge base (*i.e.*, the agent's neural network), which produces a score for the page by doing forward-propagation (Rumelhart, Hinton, and Williams 1986). Finally, the agent's neural network incorporates the user's advice and the environment's feedback, both of which affect the score of a page.

The knowledge base of a Wawa agent is centered around two basic functions: ScoreLink and ScorePage (see Figure 6). If given highly accurate instances of such functions, standard heuristic search would lead to effective retrieval of text documents: the best-scoring links would be traversed and the highest-scoring pages would be collected.

Users are able to tailor an agent's behavior by providing advice about the

[1]Portions of this chapter were previously published in Shavlik and Eliassi-Rad (1998a, 1998b), Shavlik *et al.* (1999), and Eliassi-Rad and Shavlik (2001a).

[2]For simplicity, the terms "Web page" and "document" are used interchangeably in this thesis.

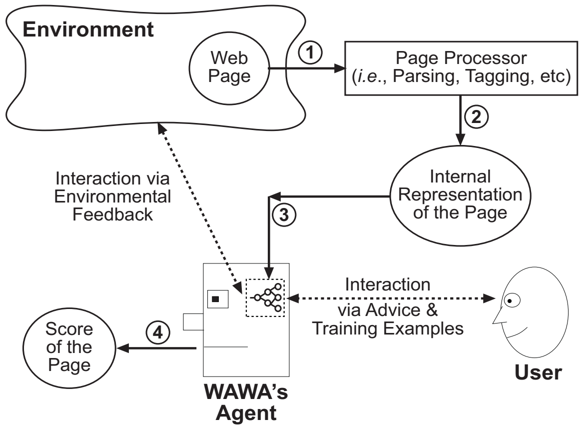

Figure 5: Scoring a Page with a WAWA Agent

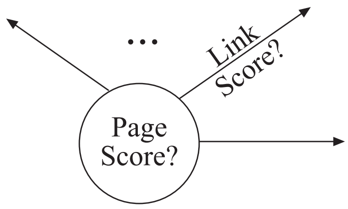

Figure 6: Central Functions of WAWA's Agents Score Web Pages and Hyperlinks

above functions. This advice is "compiled" into two "knowledge based" neural networks (see Section 3.3) implementing the functions ScoreLink and ScorePage (see Figure 7). These functions, respectively, guide the agent's wandering within the Web and judge the value of the pages encountered. Subsequent reinforcements from the Web (*e.g.*, encountering dead links) and any ratings of retrieved pages that the user wishes to provide are, respectively, used to refine the link- and page-scoring functions.

A Wawa agent's ScorePage network is a supervised learner (Mitchell 1997). That is, it learns through user-provided training examples and advice. A Wawa agent's ScoreLink network is a reinforcement learner (Sutton and Barto 1998). This network automatically creates its own training examples, though it can also use any user-provided training examples and advice. Hence,

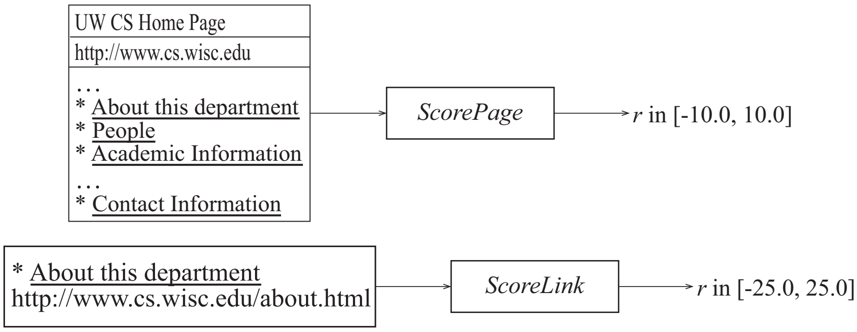

Figure 7: WAWA's Central Functions Score Web Pages and Hyperlinks

WAWA's design of the SCORELINK network has the important advantage of producing self-tuning agents since training examples are created by the agent itself (see Section 4.3).

3.1 Input Features

WAWA extracts features from either HTML or plain-text Web pages. In addition to representing WAWA's input units, these input features constitute the primitives in its advice language. This section presents WAWA's feature-extraction method.

A standard representation of text used in IR is the *bag-of-words* representation (Salton 1991). In the bag-of-words representation, word order is lost and all that is used is a vector that records the words on the page (usually scaled according to the number of occurrences and other properties; see Section 2.4). The top-right part of Figure 8 illustrates this representation.

Generally, IR systems (Belew 2000) reduce the dimensionality (*i.e.*, number of possible features) in the problem by discarding common ("stop") words and "stemming" all words to their root form (*e.g.*, "walked" becomes "walk"). WAWA implements these two preprocessing steps by using a generic list of stop words and Porter's stemmer (1980). In particular, I use the popular Frakes and Cox's implementation of Porter's stemmer (Frakes and Baeza-Yates 1992).

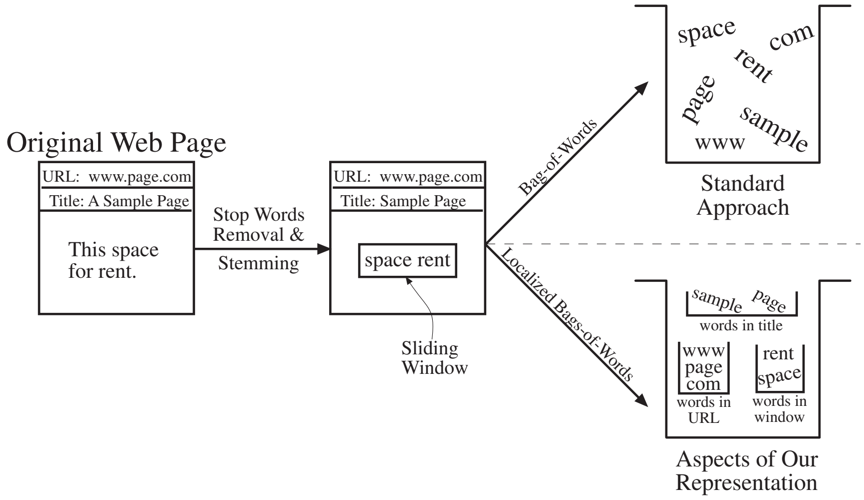

Figure 8: Internal Representation of Web Pages

The information provided by word order is usually important. For example, without word-order information, instructions such as *find pages containing the phrase "Green Bay Packers"* cannot be expressed. Given that WAWA uses neural networks to score pages and links, one approach to capturing word-order information would be to use recurrent networks (Elman 1991). However, WAWA borrows an idea from NETTALK (Sejnowski and Rosenberg 1987), though WAWA's basic unit is a word rather than an (alphabetic) letter as in NETTALK. Namely, WAWA "reads" a page by sliding a fixed-size window across a page one word at a time. Typically, the sliding window contains 15 words. Figure 9 provides an example of a three-word sliding window going across a page.

Most of the features WAWA uses to represent a page are defined with respect to the current center of the sliding window. The sliding window itself captures word order on a page; however, WAWA also maintains two bags of words (each of size 10) that surround the sliding window. These neighboring bags allows WAWA to capture instructions such as *find pages where "Green Bay" is near "Packers"*.

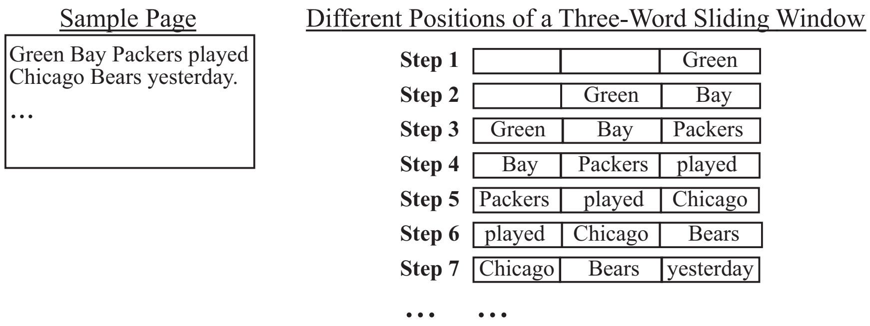

Figure 9: Using a Three-Word Sliding Window to Capture Word-Order Information on a Page

In addition to preserving some word-order information, Wawa also takes advantage of the structure of HTML documents (when a fetched page is so formatted). First, it augments the bag-of-words model, by using several localized bags, some of which are illustrated on the bottom-right part of Figure 8. Besides a bag for all the words on the page, Wawa has word bags for: the title, the URL of a page, the sliding window, the left and right sides of the sliding window, the current hyperlink[3] (should the window be inside hypertext), and the current section's title. Wawa's parser of Web pages records the "parent" section title of each word; parents of words are indicated by the standard ⟨H1⟩ through ⟨H6⟩ section-header constructs of HTML, as well as other indicators such as table captions and table-column headings. Moreover, bags for the words in the grandparent and great-grandparent sections are kept, should the current window be nested that deeply.

Wawa uses Brill's tagger (1994) to annotate each word on a page with a part-of-speech (POS) tag (*i.e.*, noun, proper noun, verb, etc). This information is represented in the agent's neural networks as input features for the words in the sliding window. By adding POS tags, Wawa is able to distinguish between

[3]A Web page has a set of URLs that point to it and a set of URLs within its contents. In this thesis, I refer to the former as *URL* and the later cases as *hyperlinks*, in an attempt to reduce confusion.

different grammatical uses of a word. For example, this allows a user to express instructions such as *find pages where the words "fly" and "bug" are nouns.* WAWA also takes advantage of the inherent hierarchy in the POS tags (*e.g.*, a proper noun is also a noun, or a present participle verb is also a verb). For example, if the user indicates interest in the word "fly" as a noun, WAWA looks for the presence of "fly" as a noun and as a proper noun. However, if the user indicates interest in the word "Bill" as a proper noun, then WAWA *only* looks for the presence of "Bill" as a proper noun and not as a noun.

Table 4 lists some of WAWA's extracted input features (see Appendix A for a full list of WAWA's input features). The features *anywhereOnPage*($\langle word \rangle$) and *anywhereInTitle*($\langle word \rangle$) take a word as input and return true if the word was on the page or inside the title of the page, respectively. These two features represent the word bags for the page and the title.

In addition to the features representing bag-of-words and word order, WAWA also represents several fixed positions. Besides the obvious case of the positions in the sliding window, WAWA represents the first and last N words (for some fixed N provided by the user) in the title, the URL, the section titles, etc. Since URLs and hyperlinks play an important role in the Web, WAWA captures the last N fields (i.e., delimited by dots) in the *server* portion of URLs and hyperlinks, e.g. `www wisc edu` in `http://www.wisc.edu/news.html`.

Besides the input features related to words and their positions on the page, a WAWA agent's input vector also includes various other features, such as the length of the page, the date the page was created or modified (should the page's server provide that information), whether the window is inside emphasized HTML text, the sizes of the various word bags, how many words mentioned in advice are present in the various bags, etc.

Features describing POS tags for the words in the sliding window are represented by the last three features in Table 4. For example, the input feature *POSatCenterOfWindow(noun)* is true only when the current word at the center

Table 4: Sample Extracted Input Features

`anywhereOnPage(`$\langle word \rangle$`)`
`anywhereInTitle(`$\langle word \rangle$`)`
...
`isNthWordInTitle(`$\langle N \rangle, \langle word \rangle$`)`
...
`isNthWordFromENDofTitle(`$\langle N \rangle, \langle word \rangle$`)`
...
`NthFromENDofURLhostname(`$\langle N \rangle, \langle word \rangle$`)`
...
`leftNwordInWindow(`$\langle N \rangle, \langle word \rangle$`)`
`centerWordInWindow(`$\langle word \rangle$`)`
...
`numberOfWordsInTitle()`
`numberOfAdviceWordsInTitle()`
...
`insideEmphasizedText()`
`timePageWasLastModified()`
...
`POSatRightSpotInWindow(`$\langle N \rangle, \langle POS\ tag \rangle$`)`
`POSatCenterOfWindow(`$\langle POS\ tag \rangle$`)`
`POSatLeftSpotInWindow(`$\langle N \rangle, \langle POS\ tag \rangle$`)`

of the sliding window is tagged as a noun. The features *POSatRightSpotInWindow* and *POSatLeftSpotInWindow* specify the desired POS tag for the Nth position to the right or left of the center of the sliding window, respectively.

WAWA's design[4] leads to a large number of input features which allows it to have an expressive advice language. For example, WAWA uses many Boolean-valued features to represent a Web page, ranging from *anywhereOnPage(aardvark)* to *anywhereOnPage(zebra)* to *rightNwordInWindow(3, AAAI)* to *NthFromENDofURLhostname(1, edu)*. Assuming a typical vocabulary of tens of thousands of words, the number of input features is on the order of a

[4]The current version of WAWA does not use any TF/IDF methods (see Section 2.4), due to the manner in which it compiles advice into networks (see Section 3.3).

million!

One might ask how a learning system could hope to do well in such a large space of input features. Fortunately, WAWA's use of advice means that users indirectly select a subset of this huge set of implicit input features. Namely, they indirectly select only those features that involve the words appearing in their advice. The full set of input features is still there, but the weights out of input features used in advice have high values, while all other weights (*i.e.*, unmentioned words and positions) have values near zero (see Section 2.2. Thus, there is the potential for words not mentioned in advice to impact a network's output, after lots of training.

WAWA also deals with the enormous input space by explicitly representing only what *is* on a page. That is, all zero-valued features, such as *anywhereOnPage(aardvark) = false*, are only *implicitly* represented. Fortunately, the nature of weighted sums in both the forward and backward propagation phases of neural networks (Rumelhart, Hinton, and Williams 1986) means that zero-valued nodes have no impact and hence can be ignored.

3.2 Advice Language

The user-provided instructions are mapped into the SCOREPAGE and SCORELINK networks using a Web-based language called *advice*. An expression in WAWA's advice language is an instruction of the following basic form:

`when` *`condition`* `then` *`action`*

The conditions represent aspects of the contents and structure of Web pages. WAWA's extracted input features (as described in Section 3.1) constitute the primitives used in the conditions of advice rules. These primitive constructs can be combined to create more complicated constructs. Table 5 lists the actions of WAWA's advice language in BNF, short for *Backus-Naur Form*, (Aho, Sethi, and Ullman 1986) notation. The `strength` levels in actions represent the degree to which the user wants to increase or decrease the score of a page or a link. Appendix A presents the advice language in its entirety.

Table 5: Permissible Actions in an Advice Statement

action →	*strength* show page
	| *strength* avoid showing page
	| *strength* follow link
	| *strength* avoid following link
	| *strength* show page & follow link
	| *strength* avoid showing page & following link
strength →	weakly | moderately | strongly | definitely

3.2.1 Complex Advice Constructs and Predicates

All features extracted from a page or a link constitute the basic constructs and predicates of the advice language.[5] These basic constructs and predicates can be combined via Boolean operators (*i.e.*, `AND, OR, NOT`) to create complex predicates.

Phrases (Croft, Turtle, and Lewis 1991), which specify desired properties of consecutive words, play a central role in creating more complex predicates out of the primitive features that WAWA extracts from Web pages. Table 6 contains some of the more complicated predicates that WAWA defines in terms of the basic input features. The advice rules in this table correspond to instructions a user might provide if she is interested in finding *Joe Smith*'s home-page.[6]

Rule 1 indicates that when the system is sliding the window across the title of a page, it should look for any of the plausible variants of *Joe Smith*'s first name, followed by his last name, apostrophe *s*, and the phrase "home page."

[5] A predicate is a function that returns either true or false. I define a construct as a function that returns numeric values.

[6] The *anyOf*`()` construct used in the table is satisfied when *any* of the listed words is present.

Table 6: Sample Advice

```
(1) WHEN consecutiveInTitle(
            anyOf(Joseph Joe J.)
            Smith's home page)
    STRONGLY SUGGEST SHOWING PAGE
(2) WHEN hyperlinkEndsWith(
              anyOf(Joseph Joe Smith jsmith)  /
              anyOf(Joseph Joe Smith jsmith
                    index home homepage my me)
              anyOf(htm html /))
    STRONGLY SUGGEST FOLLOWING LINK
(3) WHEN (titleStartsWith(Joseph Joe J.)
          and titleEndsWith(Smith))
    SUGGEST SHOWING PAGE
(4) WHEN NOT(anywhereOnPage(Smith))
    STRONGLY SUGGEST AVOID SHOWING PAGE
```

Rule 2 demonstrates another useful piece of advice for home-page finding. This one gets compiled into the *NthFromENDofHyperlink()* input features, which are true when the specified word is the *Nth* one from the *end* of the current hyperlink. (Note that WAWA treats the '/' in URLs as a separate word.)

Rule 3 depicts an interest in pages that have titles starting with any of the plausible variants of *Joe Smith*'s first name and ending with his last name.

Rule 4 shows that advice can also specify when *not* to follow a link or show a page; negations and AVOID instructions become negative weights in the neural networks.

3.2.2 Advice Variables

WAWA's advice language contains variables, which by definition range over various kinds of concepts, like names, places, etc. Advice variables are of particular relevance to WAWA's IE system (see Section 6 for details).

To understand how variables are used in WAWA, assume that a user wishes to utilize the system to create a home-page finder. She might wish to give such a system some (very good) advice like:

```
when consecutiveInTitle(?FirstName ?LastName 's Home
Page) then show page
```

The leading question marks (*?*) indicate variables that are bound upon receiving a request to find a specific person's home page. The use of variables allows the same advice to be applied to the task of finding the home pages of any number of different people. The next section further explains how variables are implemented in WAWA.

3.3 Compilation of Advice into Neural Networks

Advice is compiled into the SCOREPAGE and SCORELINK networks using a variant of the KBANN algorithm (Towell and Shavlik 1994). The mapping process (see Section 2.2) is analogous to compiling a traditional program into machine code, but WAWA instead compiles advice rules into an intermediate language expressed using neural networks. This provides the important advantage that WAWA's "machine code" can automatically be refined based on feedback provided by either the user or the Web. Namely, WAWA can apply the backpropagation algorithm (Rumelhart, Hinton, and Williams 1986) to learn from the training set.

I will illustrate the mapping of an advice rule with variables through an example. Suppose WAWA is given the following advice rule:

```
when consecutive( Professor ?FirstName ?LastName )
then show page
```

During advice compilation, WAWA maps the phrase by centering it over the sliding window (Figure 10). In this example, the phrase is a sequence of three words, so it maps to three positions in the input units corresponding to the sliding window (with the variable *?FirstName* associated with the center of the sliding window).

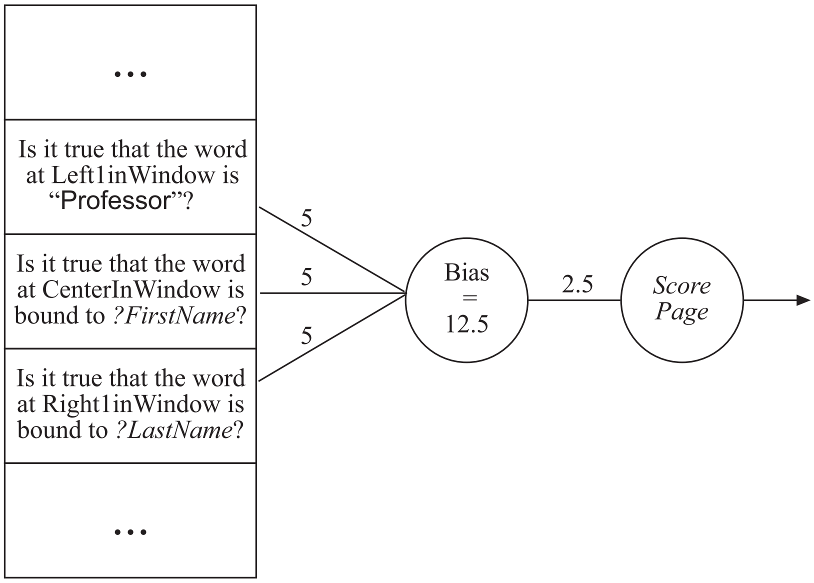

Figure 10: Mapping Advice into ScorePage Network

The variables in the input units are bound outside of the network and the units are turned on only if there is a match between the bindings and the words in the current position of the sliding window. Assume the bindings are:

$$?FirstName \leftarrow \text{“Joe”}$$
$$?LastName \leftarrow \text{“Smith”}$$

Then, the input unit "*Is it true that the word CenterInWindow is bound to ?FirstName*?" will be true (*i.e.*, set to 1) only if the current word in the center of the window is "Joe." Similarly the input unit "*Is it true that the Right1inWindow is bound to ?LastName*?" will be set to 1 only if the current word immediately to the right of the center of the window is "Smith."

Wawa then connects the referenced input units to a newly created hidden unit, using weights of value 5. Next, the bias (*i.e.*, the threshold) of the new hidden unit is set such that all the required predicates must be true in order for the weighted sum of its inputs to exceed the bias and produce an activation of the sigmoidal hidden unit near 1. Some additional zero-weighted links are also added to this new hidden unit, to further allow subsequent learning, as is standard in Kbann.

Finally, Wawa links the hidden unit into the output unit with a weight determined by the strength given in the rule's action. Wawa interprets the action *show page* as "moderately increase the page's score."

The mapping of advice rules without variables follows the same process except that there is no variable-binding step.

3.4 Scoring Arbitrarily Long Pages and Links

Wawa's use of neural networks means that it needs a mechanism for processing arbitrarily long Web pages with the fixed-sized input vectors used by neural networks. Wawa's sliding window resolves this problem. Recall that the sliding window extracts features from a page by moving across it one word at a time. There are, however, some HTML tags like ⟨P⟩, ⟨/P⟩, ⟨BR⟩, and ⟨HR⟩ that act as "window breakers." Window breakers[7] do not allow the sliding window to cross over them because such markers indicate a new topic. When a window breaker is encountered, the unused positions in the sliding window are left unfilled.

The score of a page is computed in two stages. In stage one, Wawa sets the input units that represent global features of the page, such as the number of words on the page. Then, Wawa slides the window (hence, the name *sliding window*) across the page. For each window position, Wawa first sets the values for the input units representing positions in the window (*e.g.*, word at center of window) and then calculates the values for all hidden units (HUs) that are *directly* connected to input units. These HUs are called "level-one" HUs. In

[7]Wawa considers the following tags as *window breakers*: "⟨P⟩," "⟨/P⟩," "⟨BR⟩," "⟨HR⟩," "⟨BLOCKQUOTE⟩," "⟨/BLOCKQUOTE⟩," "⟨PRE⟩," "⟨/PRE⟩," "⟨XMP⟩," and "⟨/XMP⟩."

other words, WAWA performs forward-propagation from the input units to all level-one HUs. This process gives WAWA a list of values for all the level-one HUs at each position of the sliding window. For each level-one HU, WAWA picks the *highest* value to represent its activation.

In stage two, the highest values of level-one HUs and the values of input units for global features are used to compute the values for all other HUs and the output unit. That is, WAWA performs forward-propagation from the level-one HUs and the "global" input units to the output unit (which obviously will evaluate the values of all other HUs in the process). The value produced by the SCOREPAGE network in the second stage is returned as the page's score.

Note that WAWA's two-phase forward-propagation process means that HUs cannot connect to both input units and other HUs. The compilation process ensures this by adding "dummy" HUs in necessary locations.

Although the score of a page is computed in two stages, WAWA scans the sliding window only once across the page, which occurs in stage one. By forward-propagating only to level-one HUs in the first stage, WAWA is effectively trying to get the values of its complex features. In stage two, WAWA uses these values and the global input units' values to find the score of the page. For example, the two-stage process allows WAWA to capture advice such as

```
when ( consecutive(Milwaukee Brewers) and
       consecutive(Chicago Cubs) ) then show page
```

If WAWA only had a one-stage process, it would not be able to correctly capture this advice rule because both phrases cannot be in the sliding window simultaneously. Figure 11 illustrates this point.

The value of a hyperlink is computed similarly, except that the SCORELINK network is used and the sliding window is slid over the *hypertext* associated with that hyperlink and the 15 words surrounding the hypertext on both sides.

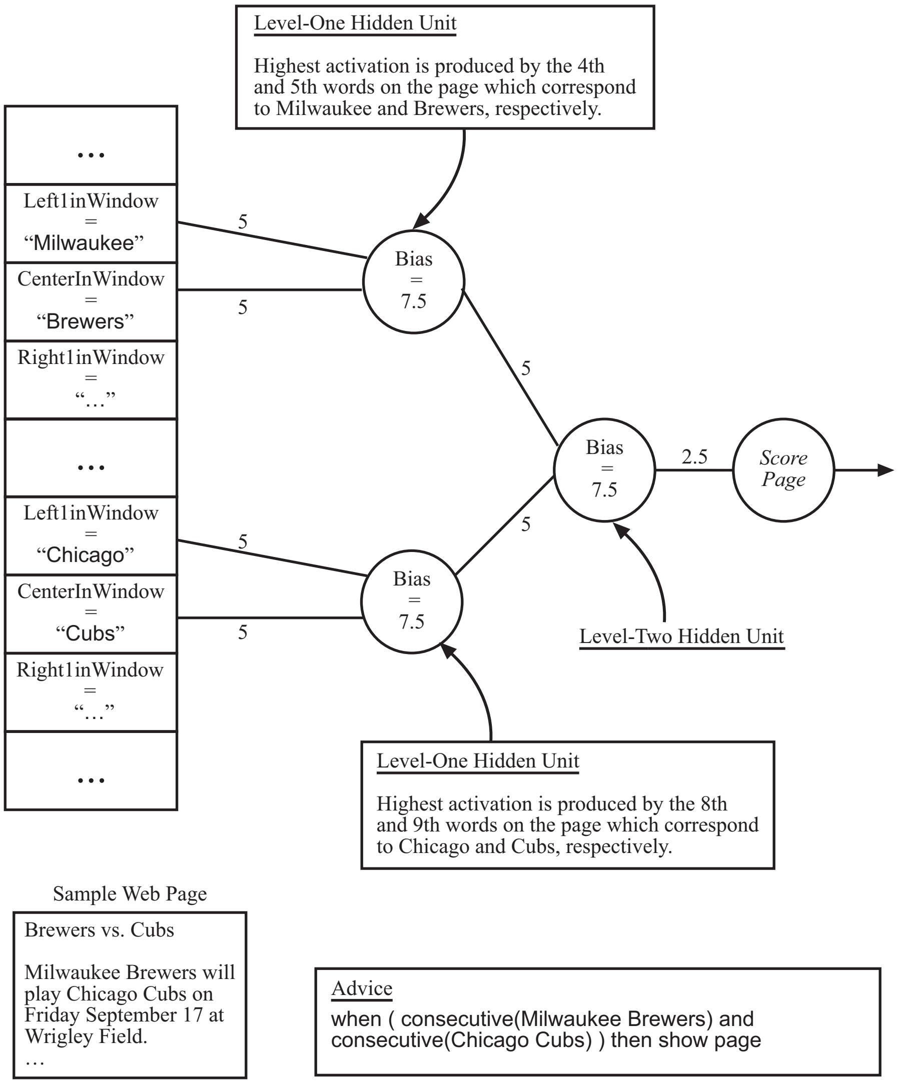

Figure 11: Scoring a Page with a Sliding Window. In stage one, the sliding window is scanned through the page to determine the highest values of the level-one hidden units. In stage two, the highest activations of the level-one hidden units are used to score the level-two hidden unit and the output unit, which produces the overall score of the page.

Chapter 4

Using WAWA to Retrieve Information from the Web

Given a set of documents (*a.k.a.* the corpus) and a query, which usually consists of a bunch of keywords or keyphrases, an "ideal" IR system is supposed to return *all and only* the documents that are relevant to the user's query. Unfortunately with the recent exponential growth of on-line information, it is almost impossible to find such an "ideal" IR system (Lawrence and Giles 1999). In an effort to improve on the performance of existing IR systems, there has been a lot of interest in using machine learning techniques to solve the IR problem (Drummond, Ionescu, and Holte 1995; Pazzani, Muramatsu, and Billsus 1996; Joachims, Freitag, and Mitchell 1997; Rennie and McCallum 1999). An *IR learner* attempts to model a user's preferences and return on-line documents "matching" those interests.

This chapter[1] describes the design of WAWA's IR learner (namely WAWA-IR), the different ways its two neural networks are trained, and the manner in which it automatically derives training examples.

4.1 IR System Description

WAWA-IR is a general search engine agent that through training can be specialized and personalized. Table 7 provides a high-level description of WAWA-IR.

[1]Portions of this chapter were previously published in Shavlik and Eliassi-Rad (1998a, 1998b), Shavlik *et al.* (1999), and Eliassi-Rad and Shavlik (2001a).

Table 7: WAWA's Information-Retrieval Algorithm

Unless they have been saved to disk in a previous session, create the *ScoreLink* and *ScorePage* neural networks by reading the user's initial advice, if any was provided (see Section 3.3).
Either (a) start by adding user-provided URLs to the search queue; or (b) initialize the search queue with URLs that will query the user's chosen set of Web search-engine sites.
Execute the following concurrent processes.
Process #1 While the search queue is not empty nor the maximum number of URLs have been visited,
Let $URLtoVisit$ = pop(search queue).   Fetch $URLtoVisit$.

Evaluate $URLtoVisit$ using $ScorePage$ network.
  If score is high enough, insert $URLtoVisit$
    into the sorted list of best pages found.
  Use the score of $URLtoVisit$ to improve
    the predictions of the $ScoreLink$ network
    (see Section 4.3 for details).

Evaluate the hyperlinks in $URLtoVisit$
    using $ScoreLink$ network (however, only
    score those links that have not yet been
    followed this session).
  Insert these new URLs into the (sorted) search
    queue if they fit within its max-length bound.

Process #2 Whenever the user provides additional advice, insert it into the appropriate neural network.
Process #3 Whenever the person rates a fetched page, use this rating to create a training example for the $ScorePage$ neural network.

Initially, WAWA-IR's two neural networks are created by either using the techniques described in Section 3 or by reading them from disk (should this be a resumption of a previous session). Then, the basic operation of WAWA-IR is heuristic search, with its SCORELINK network acting as the heuristic function. Rather than solely finding one goal node, WAWA-IR collects the 100 pages that SCOREPAGE rates highest. The user can choose to seed the queue of pages to fetch in two ways: either by specifying a set of starting URLs or by providing a simple query that WAWA-IR converts into "query" URLs that are sent to a user-chosen subset of selectable search engine sites (currently ALTAVISTA, EXCITE, GOOGLE, HOTBOT, INFOSEEK, LYCOS, TEOMA,[2] WEBCRAWLER, and YAHOO).

Although not mentioned in Table 7, the *user* may also specify values for the following parameters:

- an upper bound on the distance the agent can wander from the initial URLs, where distance is defined as the number hyperlinks followed from the initial URL (default value is 10)
- minimum score a hyperlink must receive in order to be put in the search queue (default value is 0.6 on a scale of [0,1])
- maximum number of hyperlinks to add from a page (default value is 50)
- maximum kilobytes to read from each page (default value is 100 kilobytes)
- maximum retrieval time per page (default value is 90 seconds).

[2]TEOMA is a new search engine which received rave reviews in the Internet Scout Report of June 21, 2001 (`http://scout.cs.wisc.edu`).

4.2 Training WAWA-IR's Two Neural Networks

There are three ways to train Wawa-IR's two neural networks: (*i*) system-generated training examples, (*ii*) advice from the user, and (*iii*) user-generated training examples.

Before fetching a page P, Wawa-IR predicts the value of retrieving P by using the ScoreLink network. This "predicted" value of P is based on the text surrounding the hyperlink to P and some global information on the "referring" page (*e.g.*, the title, the URL, etc). After fetching and analyzing the actual text of P, Wawa-IR re-estimates the value of P, this time using the ScorePage network. Any differences between the "before" and "after" estimates of P's score constitute an error that can be used by backpropagation (Rumelhart, Hinton, and Williams 1986) to improve the ScoreLink neural network. The details of this process are further described in Section 4.3. This type of training is *not* performed on the pages that constitute the initial search queue because their values were not predicted by the ScoreLink network.

In addition to the above system-internal method of automatically creating training examples, the user can improve the ScorePage and ScoreLink neural networks in two ways: (*i*) by providing additional advice and (*ii*) by providing training examples. Observing the agent's behavior is likely to invoke thoughts of good additional instructions (as has repeatedly happened to me in my experiments). A Wawa-IR agent can accept new advice and augment its neural networks at any time. It simply adds to its networks additional hidden units that represent the compiled advice, a technique whose effectiveness was previously demonstrated on several tasks (Maclin and Shavlik 1996). Providing additional hints can rapidly and drastically improve the performance of a Wawa-IR agent, provided the advice is relevant. Maclin and Shavlik (1996) showed that their algorithm is robust when given advice incrementally. When "bad" advice was given, the agent was able to quickly learn to ignore it.

Although more tedious, the user can also rate pages as a mechanism for providing training examples for use by backpropagation. This can be useful when the user is unable to articulate why the agent is misscoring pages and links. This standard learning-from-labeled-examples methodology has been previously investigated by other researchers, *e.g.*, Pazzani *et al.* (1996), and this aspect of Wawa-IR is discussed in Section 5. However, I conjecture that most of the improvement to Wawa-IR's neural networks, especially to ScorePage, will result from users providing advice. In my personal experience, it is easy to think of simple advice that would require a large number of labeled examples in order to learn purely inductively. In other words, one advice rule typically covers a large number of labeled examples. For example, a rule such as

```
when consecutive(404 file not found) then avoid showing page
```

will cover all pages that contain the phrase "404 file not found."

4.3 Deriving Training Examples for ScoreLink

Wawa-IR uses *temporal-difference* methods (Sutton 1988) to automatically train the ScoreLink network. Specifically, it employs a form of Q-learning (Watkins 1989), which is a type of reinforcement learning (Sutton and Barto 1998). Recall that the difference between Wawa-IR's prediction of the link's value before fetching a url and its new estimate serves as an error that backpropagation tries to reduce. Whenever Wawa-IR has collected all the necessary information to re-estimate a link's value, it invokes backpropagation. In addition, it periodically reuses these training examples several times to refine the network. The main advantage of using reinforcement learning to train the ScoreLink network is that Wawa-IR is able to *automatically* construct these training examples without direct user intervention.

As is typical in reinforcement learning, the value of an action (following a hyperlink in this case) is *not* solely determined by the immediate result of the action (*i.e.*, the value of the page retrieved minus any retrieval-time penalty). Rather, it is important to also reward links that *lead to* pages with additional good links on them. Figure 12 and Equation 1 illustrate this point.

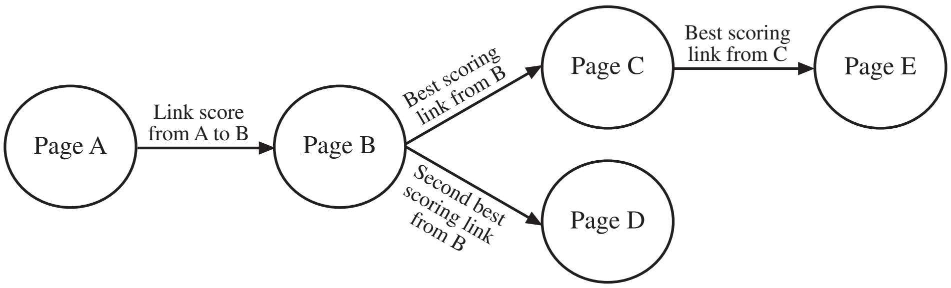

Figure 12: Reestimating the Value of a Hyperlink

Equation 1: New Estimate of the Link $A \to B$ under Best-First Search

$$
\begin{array}{l}
\texttt{if } ScoreLink(B \to C) > 0 \texttt{ then} \\
\quad \text{new estimate of } ScoreLink(A \to B) \\
\qquad = fetchPenalty(B) + ScorePage(B) \\
\qquad\quad + \gamma(fetchPenalty(C) + ScorePage(C)) \\
\qquad\quad + \gamma^2 \text{MAX}(0, ScoreLink(B \to D), \\
\qquad\qquad\qquad\qquad ScoreLink(C \to E)) \\
\texttt{else} \\
\quad \text{new estimate of } ScoreLink(A \to B) \\
\qquad = fetchPenalty(B) + ScorePage(B)
\end{array}
$$

WAWA-IR defines the task of the SCORELINK function to be estimating the discounted sum of the scores of the fetched pages plus the cost of fetching them. The *discount rate*, γ in Equation 1, determines the amount by which the value of a page is discounted because it is encountered at a later time step. γ has a default value of 0.95 (where $\gamma \in [0, 1]$). The closer the value of γ is to 1, the more strongly the values of future pages are taken into account. The cost of fetching a page depends on its size and retrieval rate.[3]

I assume that *the system started its best-first search at the page referred to by the hyperlink*. In other words, if in Figure 12, Page B was the root of a best-first search, WAWA-IR would next visit C and then either D or E, depending on which referring hyperlink scored higher. Hence, the first few terms of the sum would be the value of root page B, plus the value of C discounted by one time step. WAWA-IR then recursively estimates the remainder of this sum by using the *discounted*[4] higher score of the two URLS that would be at the front of the search queue.

Since WAWA-IR uses best-first search, it actually may have a more promising URL in its search queue than the link to C (assuming the move from A to B took place). In order to keep the calculation of the re-estimated SCORELINK function localized, I largely ignore this aspect of the system's behavior. WAWA-IR only partially captures this phenomenon by adjusting the above calculation such that the links with negative predicted value are not followed.[5]

The above scenario (of localizing the re-estimation of SCORELINK function) does not apply when an URL cannot be fetched (*i.e.*, a "dead link"). Upon such an occurrence, SCORELINK receives a large penalty. Depending on the type of failure, the default values range from -6.25 (for failures such as "file not found") to 0 (for failures like "network was down").

The definition of WAWA-IR's "Q-function" (see Equation 1) represents a best-first, beam-search strategy. This definition is different from the traditional definition (see Equation 2), which essentially assumes a hill-climbing approach.

[3]The cost of fetching a page is defined to be $(-1.25 \times rate) + (-0.001 \times rate \times size)$ and has a maximum value of -4.85.

[4]The score is doubly discounted here since the URLS are two time steps in the future.

[5]WAWA-IR's networks are able to produce negative values because their output units simply output their weighted sum of inputs, *i.e.*, they are linear units. Note that the hidden units in WAWA-IR's networks use sigmoidal activation functions.

Equation 2: New Estimate of the Link $A \rightarrow B$ under Hill-Climbing Search

$$
\begin{array}{l}
\texttt{if } ScoreLink(B \rightarrow C) > 0 \texttt{ then} \\
\quad \text{new estimate of } ScoreLink(A \rightarrow B) \\
\quad\quad = fetchPenalty(B) + ScorePage(B) \\
\quad\quad\quad + \gamma(fetchPenalty(C) + ScorePage(C)) \\
\quad\quad\quad + \gamma^2 \mathrm{MAX}(0,\, ScoreLink(C \rightarrow E)) \\
\texttt{else} \\
\quad \text{new estimate of } ScoreLink(A \rightarrow B) \\
\quad\quad = fetchPenalty(B) + ScorePage(B)
\end{array}
$$

Reviewing Figure 12, one can understand the difference between WAWA-IR's approach and the traditional way of defining the Q-function. In the traditional (*i.e.*, hill-climbing) approach, since $B \rightarrow D$ was the *second* best-scoring link from B, its value is not reconsidered in the calculation of the score of $A \rightarrow B$. This search strategy does not seem optimal for finding the most relevant pages on the Web. Instead, the link with the highest-score from the set of encountered links should always be traversed. For example, if an agent has to choose between the links $B \rightarrow D$ and $C \rightarrow E$, it should follow the link that has the highest score and not ignore $B \rightarrow D$ simply because this link was seen at a previous step and did not have the highest value at that step.

4.4 Summary

The main advantage of WAWA's IR system is its use of theory refinement. That is, WAWA utilizes the user's prior knowledge, which need not be perfectly correct to guide WAWA-IR. WAWA-IR is a learning system, so it is able to improve the user's instructions. I am able to rapidly transform WAWA-IR, which is a general search engine, into a specialized and personalized IR agent by merely adding a simple "front-end" user interface that accepts domain-specific information and uses it to create rules in WAWA's advice language. Chapter 5 describes the rapid creation of an effective "home-page finder" agent from the generic WAWA-IR system.

I also allow the user to continually provide advice to the agent. This characteristic of WAWA-IR enables the user to observe an agent and guide its behavior (whenever the user feels that WAWA-IR agent's user model is incorrect). Finally, by learning the SCORELINK function, a WAWA-IR agent is able to more effectively search the Web (by learning about relevant links) *and* automatically create its own training examples via reinforcement learning (which in turn improves the accuracy of the agent with respect to the relevancy of the pages returned).

Due to my use of artificial neural networks, it is difficult to understand what was learned (Craven and Shavlik 1996). It would be nice if a WAWA-IR agent could explain its reasoning to the user. In an attempt to alleviate this problem, WAWA-IR has a "visualizer" for each of its two neural networks (Craven and Shavlik 1992). The visualizer draws the neural networks containing the user's compiled advice and graphically displays information on all nodes and links in the network.

Chapter 5

Retrieval Experiments with WAWA

This chapter[1] describes a case study done in 1998 and repeated in 2001 to evaluate WAWA's IR system.[2] I built a home-page-finder agent by using WAWA's advice language. Appendix B presents the complete advice used for the home-page finder. The results of this empirical study illustrate that by utilizing WAWA-IR, a user can build an effective agent for a web-based task quickly.

5.1 An Instructable and Adaptive Home-Page Finder

In 1998, I chose the task of building a home-page finder because of an existing system named AHOY! (Shakes, Langheinrich, and Etzioni 1997), which provided a valuable benchmark. Ahoy! uses a technique called Dynamic Reference Sifting, which filters the output of several Web indices and generates new guesses for URLs when no promising candidates are found.

I wrote a simple interface layered on top of WAWA-IR (see Figure 13) that asks for whatever relevant information is known about the person whose home page is being sought: first name, possible nicknames, middle name or initial, last

[1]Portions of this chapter were previously published in Shavlik and Eliassi-Rad (1998a, 1998b), Shavlik *et al.* (1999), and Eliassi-Rad and Shavlik (2001a).

[2]I reran the experiment in 2001 to compare WAWA-IR's performance to GOOGLE, which did not exist in 1998, and also to measure the sensitivity of some of the design choices made in the 1998 experiments.

name, miscellaneous phrases, and a partial URL (e.g., `edu` or `ibm.com`). I then wrote a small program that reads these fields and creates advice that is sent to WAWA-IR. I also wrote 76 general advice rules related to home-page finding, many of which are slight variants of others (e.g., with and without middle names or initials). Specializing WAWA-IR for this task and creating the initial general advice took only one day, plus I spent parts of another 2-3 days tinkering with the advice using 100 examples of a "training set" (described below). This step allowed me to manually refine my advice – a process which I expect will be typical of future users of WAWA-IR.

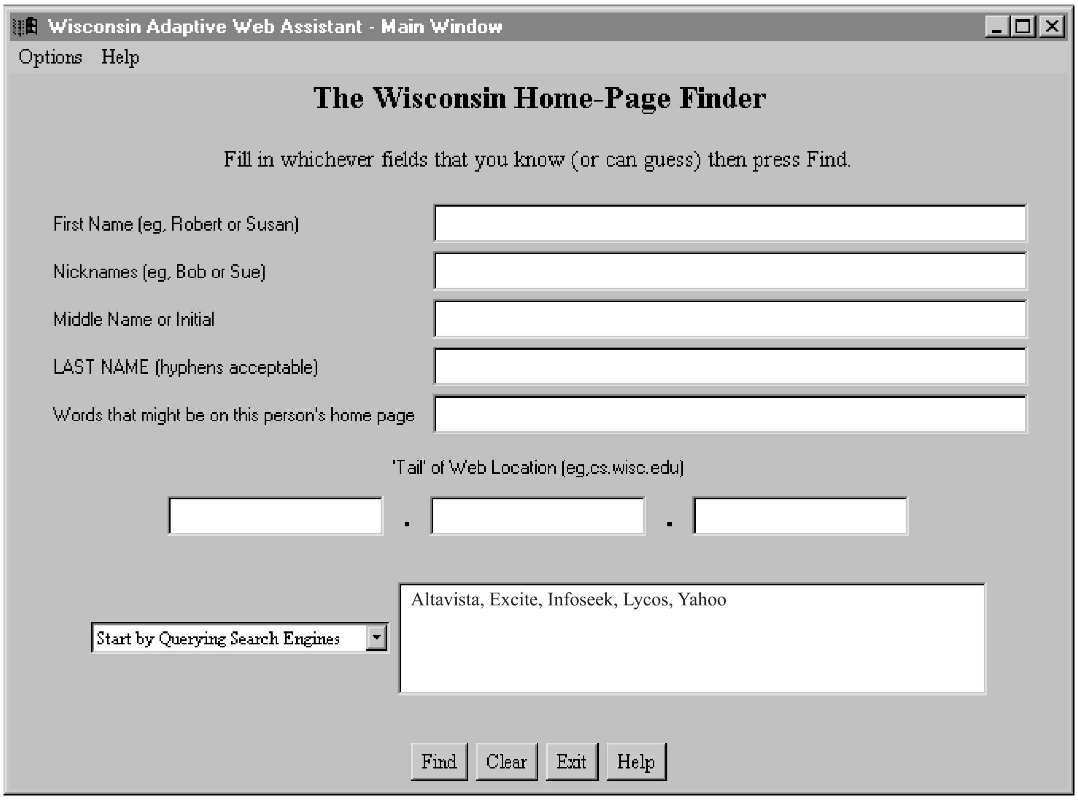

Figure 13: Interface of WAWA-IR's Home-Page Finder

To learn a general concept about home-page finding, I used the *variable binding* mechanism of WAWA-IR's advice language. WAWA-IR's home-page finder accepts instructions that certain words should be bound to variables associated with first names, last names, etc. I wrote general-purpose advice about home-page finding that uses these variables. Hence, rule 1 in Table 6 of Chapter 3 is actually written using advice variables (as illustrated in Figure 10 of Chapter 3) and not the names of specific people.

Since the current implementation of WAWA-IR can refer only to advice variables when they appear in the sliding window, advice that refers to other aspects of a Web page needs to be specially created and subsequently retracted for each request to find a specific person's home page.[3] The number of these specific-person rules that WAWA-IR's home-page finder creates depends on how much information is provided about the target person. For the experiments below, I only provide information about people's names so that the home-page finder would be as generic as possible. This leads to the generation of one to two dozen rules, depending on whether or not middle names or initials are provided.

5.2 Motivation and Methodology

In 1998, I randomly selected 215 people from Aha's list of machine learning (ML) and case-based reasoning (CBR) researchers (`www.aic.nrl.navy.mil/~aha/people.html`) to run experiments that evaluate WAWA-IR. Out of the 215 people selected, I randomly picked 115 of them to train WAWA-IR and used the remaining 100 as my test set.[4] In 2001, I updated the data set by replacing the people that no longer had personal home pages with randomly selected people from the 2001 version of Aha's list.

The "training" phase has two steps. In the first step, I manually run the system on 100 people randomly picked from the training set (I will refer to this

[3]Users can retract advice from WAWA-IR's neural networks. To retract an advice rule, WAWA-IR removes the network nodes and links associated with that rule.

[4]I follow standard machine learning methodology in dividing the data set into two subsets, where one subset is used for training and the other for testing purposes.

set as the *advice-training set*), refining my advice by hand before "freezing" the advice-giving phase. In the second step, I split the remaining 15 people into a set of 10 people for backpropagation training (I will refer to this set as the *machine-training set*) and a set consisting of 5 people for tuning (I will refer to this set as the *tuning set*).

I do not perform backpropagation-based training during the first step of the advice-training phase, since I want to see how accurate I can make my advice without any machine learning. Then, for each person in the machine-training set, Wawa-IR initiates a search (by using the designated search engines) to find training pages and links. The ScorePage function is then trained via backpropagation. The tuning set is used to avoid overfitting the training examples.[5] For refining the ScoreLink function, Wawa-IR automatically generates training examples for each person via temporal-difference learning (see Section 4.3). Finally, I evaluate Wawa-IR's "trained" home-page finder on the test set. During the testing phase, no learning takes place.

I consider one person as one training example, even though for each person I rate several pages. Hence, the actual number of different input-output pairs processed by backpropagation is larger than the size of my training set (*i.e.*, by a factor of about 10). Table 8 describes my technique.

Table 8: The Supervised-Learning Technique Used in Training **ScorePage** Network. See text for explanation of desired output values used during training.

While the error on the tuning set is *not* increasing do the following:
For each person in the machine-training set do the following 10 times:
If the person's home page was found,
then train the ScorePage network on those pages that
scored higher than the home page, the actual home page, and
the five pages that scored immediately below the actual home page.
Otherwise, train the network on the 5 highest-scoring pages.
Calculate the error on the tuning set.

[5]When a network is overfit, it performs very well on training data but poorly on new data.

To do neural-network learning, I need to associate a desired score to each page WAWA-IR encounters. I will then be able to compare this score to the output of the SCOREPAGE network for this page and finally perform error back-propagation (Rumelhart, Hinton, and Williams 1986). I use a simple heuristic for getting the desired score of a page. I define a *target page* to be the actual home page of a person. Recall that the score of a page is a real number in the interval [-10.0, 10.0]. My heuristic is as follows:

- If the page encountered is the target page, its desired score is 9.5.
- If the page encountered has the same host as the target page, its desired score is 7.0.
- Otherwise, the desired score of the page encountered is -1.0.

For example, suppose the target person is “Alan Turing” and the target page is `http://www.turing.org.uk/turing/` (*i.e.*, his home page). Upon encountering a page at `http://www.turing.org.uk/`, I will set its desired score to 7.0, since that page has the same host as Alan Turing’s home page.

In the 2001 experiments, I compare the sensitivity of WAWA-IR’s homepage finder to the above heuristic by changing the desired score of pages with the same host as the target page from 7.0 to 3.0.[6]

To judge WAWA-IR’s performance in the task of finding home-pages, I provide it with the advice discussed above and presented in Appendix B. It is important to note that *for this experiment* I intentionally do *not* provide advice that is specific to ML, CBR, AI research, etc. By doing this, I am able to build a generalized home-page finder and not one that specializes in finding ML, CBR, and AI researchers. WAWA-IR has several options, which affect its performance both in the amount of execution time and the accuracy of its results. Before running any experiments, I choose small numbers for my parameters, using 100

[6]It turns out (see Section 5.4 that 3.0 works slightly better.

for the maximum number of pages fetched, and 3 as the maximum distance to travel away from the pages returned by the search engines.

I start WAWA-IR by providing it the person's name as given on Aha's Web page, though I partially standardize my examples by using all common variants of first names (e.g., "Joseph" and "Joe"). WAWA-IR then converts the name into an initial query (see the next paragraph). For the 1998 experiments, this initial query was sent to the following five search engines: ALTAVISTA, EXCITE, INFOSEEK, LYCOS, and YAHOO.

In my 1998 experiments, I compared the performance of WAWA-IR with the performances of AHOY! and HOTBOT, a search engine not used by WAWA-IR and the one that performed best in the home-page experiments of Shakes et al. (1997). I provided the names in my test set to AHOY! via its Web interface. Ahoy! uses *MetaCrawler* as its search engine, which queries nine search engines as opposed to WAWA-IR, which queried only five search engines in 1998. I ran HOTBOT under two different conditions. The first setting performed a specialized HOTBOT search for people; I used the names given on Aha's page for these queries. In the second variant, I provided HOTBOT with a general-purpose disjunctive query, which contained the person's last name as a *required* word, and all the likely variants of the person's first name. The latter was the same query that WAWA-IR initially sends to the five search engines used in 1998. For my experiments, I only looked at the first 100 pages that HOTBOT returned and assumed that few people would look further into the results returned by a search engine.

In my 2001 experiments, I compared the performances of different WAWA-IR settings with the performance of GOOGLE. I ran WAWA-IR with two different sets of search engines: (*i*) ALTAVISTA, EXCITE, INFOSEEK, LYCOS, TEOMA and (*ii*) GOOGLE. I did not use YAHOO, WEBCRAWLER, and HOTBOT in my 2001 experiments since they are powered by GOOGLE, EXCITE, and LYCOS, respectively. Since GOOGLE's query language does not allow me to express, *in one attempt*, the same general-purpose disjunctive query as the one described in the last paragraph, I broke that query down into the following queries:

1. "*?FirstName ?LastName*" *?FirstName +?LastName*
2. "*?NickName ?LastName*" *?NickName +?LastName*
3. "*?FirstName ?LastName*" *?FirstName +?LastName* "home page"
4. "*?NickName ?LastName*" *?NickName +?LastName* "home page"
5. "*?FirstName ?LastName*" *?FirstName +?LastName* home-page
6. "*?NickName ?LastName*" *?NickName +?LastName* home-page

The variables *?FirstName*, *?NickName*, and *?LastName* get bound to each person's first name, nick name (if one is available), and last name, respectively. Each query asks for pages that have all the terms of the query on the page. For each person, I examine the first 100 pages that GOOGLE returns and report the highest overall rank of a person's home page (if the target page was found). I send the same queries to the search engines that seed WAWA-IR's home-page finder.

Since people often have different URLS pointing to their home pages, rather than comparing URLS to those provided on Aha's page, I instead do an exact comparison on the contents of fetched pages to the contents of the page linked to Aha's site. Also, when running WAWA-IR, I never fetch any URLS whose server matched that of Aha's page, thereby preventing WAWA-IR from visiting Aha's site.

5.3 Results and Discussion from 1998

Table 9 lists the best performance of WAWA-IR's home-page finder and the results from AHOY! and HOTBOT. *SL* and *RL* are used to refer to supervised and reinforcement learning, respectively. Recall that, SL is used to train

ScorePage and RL to train ScoreLink. Besides reporting the percentage of the 100 test set home-pages found, I report the average ordinal position (*i.e.*, rank) *given that a page is found*, since Wawa-IR, Ahoy!, and HotBot all return sorted lists.

Table 9: Empirical Results: Wawa-IR *vs* Ahoy! and HotBot (SL = Supervised Learning and RL = Reinforcement Learning)

System	**% Found**	**Mean Rank Given Page Found**
Wawa-IR with SL, RL, & 76 rules	92%	1.3
Ahoy!	79%	1.4
HotBot person search	66%	12.0
HotBot general search	44%	15.4

These results provide strong evidence that the version of Wawa-IR, specialized into a home-page finder by adding simple advice, produces a better home-page finder than does the proprietary people-finder created by HotBot or by Ahoy!. The difference (in percentage of home-pages found) between Wawa-IR and HotBot in this experiment is statistically significant at the 99% confidence level. The difference between Wawa-IR and Ahoy! is statistically significant at the 90% confidence level. Recall that I specialize Wawa-IR's generic IR system for this task in only a few days.

Table 10 lists the home-page finder's performance without supervised and/or reinforcement learning. The motivation is to see if I gain performance through learning. I also remove 28 of my initial 76 rules to see how much my performance degrades with less advice. The 28 rules removed refer to words one might find in a home page that are not the person's name (such as "resume", "cv", "phone", "address", "email", etc). Table 10 also reports the performance of Wawa-IR when its home-page finder is trained with less than 76 advice rules and/or is not trained with supervised or reinforcement learning.

The differences between the Wawa-IR runs containing 76 advice rules with learning and without learning are not statistically significant. When I reduce

Table 10: Empirical Results on Different Versions of WAWA-IR's Home-Page Finder (SL = Supervised Learning and RL = Reinforcement Learning)

SL	**RL**	**# of Advice Rules**	**% Found**	**Mean Rank Given Page Found**
⋆	⋆	76	92%	1.3
	⋆	76	91%	1.2
		76	90%	1.6
⋆	⋆	48	89%	1.2
	⋆	48	85%	1.4
		48	83%	1.3

the number of advice rules, WAWA-IR's performance deteriorates. The results show that WAWA-IR is able to learn and increase its accuracy by 6 percentage points (from 83% with no learning to 89% with both supervised and reinforcement learning); however, the difference is not statistically significant at the 90% confidence level. It is not surprising that WAWA-IR is not able to reach its best performance, since I do not increase the size of my training data to compensate for the reduction in advice rules. Nonetheless, even with 48 rules, the difference in percentage of home pages found by WAWA-IR and by HOTBOT (in this experiment) is statistically significant at the 95% confidence level.

In the cases where the target page for a specific person is found, the mean rank of the target page is similar in all runs. Recall that the mean rank of the target page refers to its ordinal position in the list of pages returned to the user. The mean rank can be lower with the runs that included some training since without training the target page might not get as high of a score as it would with a trained network.

Assuming that WAWA-IR finds a home page, Table 11 lists the average number of pages fetched before the actual home page. Learning reduces the number of pages fetched before the target page is found. This is quite intuitive. With more learning, WAWA-IR is able to classify pages better and find the target page quicker. However, in Table 11, the average number of pages fetched (before the target page is found) is lower with 48 advice rules than with 76 advice rules. For example, with SL, RL, and 76 rules, the average is 22. With SL, RL, and 48 rules, the average is 15. At first glance, this might not seem intuitive. The reason for this discrepancy can be found in the 28 advice rules that I take out. Recall that these 28 advice rules improve the agent's accuracy by refering to words one *might* find in a home page that are *not* the person's name (such as "resume", "cv", "phone", "address", "email", etc). With these rules, the ScorePage and ScoreLink networks rate more pages and links as "promising" even though they are not home pages. Hence, more pages are fetched and processed before the target page is found.

Table 11: Average Number of Pages Fetched by Wawa-IR Before the Target Home Page (SL = Supervised Learning and RL = Reinforcement Learning)

SL	**RL**	**# of Advice Rules**	**Avg Pages Fetched Before Home Page**
⋆	⋆	76	22
	⋆	76	23
		76	31
⋆	⋆	48	15
	⋆	48	17
		48	24

5.4 Results and Discussion from 2001

Table 12 compares the best performances of Wawa-IR's home-page finder seeded with and without Google to the results from Google (run by itself). The Wawa-IR run seeded without Google uses the following search engines: AltaVista, Excite, InfoSeek, Lycos, and Teoma. For the runs reported in Table 12, I trained the Wawa-IR agent with reinforcement learning, supervised learning,[7] and all 76 home-page finding advice rules.

[7] I used a target score of 3.0 for the pages that had the same host as the target home page.

Table 12: Two Different WAWA-IR Home-Page Finders versus GOOGLE

System	% Found	Mean Rank±Variance Given Page Was Found
WAWA-IR with GOOGLE	96%	1.12± 0.15
GOOGLE	95%	2.01±16.64
WAWA-IR without GOOGLE	91%	1.14± 0.15

WAWA-IR seeded with GOOGLE is able to slightly improve on GOOGLE's performance by finding 96 of the 100 pages in the test set. Wawa-IR seeded without GOOGLE is not able to find more home pages than GOOGLE. This is due to the fact that the aggregate of the five search engines used is not as accurate as GOOGLE. In particular, GOOGLE appears to be quite good at finding home pages due to its *PageRank* scoring function, which globally ranks a Web page based on its location in the Web's graph structure and not on the page's content (Brin and Page 1998).

It is interesting to note that WAWA-IR and GOOGLE only share one page among the list of pages that they both do not find. The four pages that WAWA-IR missed belong to people with very common names. For example, GOOGLE is able to return *Charles "Chuck" Anderson*'s home page. But WAWA-IR fills its queue with home pages of other people or companies named *Charles "Chuck" Anderson*.[8] On the other hand, the pages that GOOGLE does not find are the ones that WAWA-IR is able to discover by following links out of the original pages returned by GOOGLE.

WAWA-IR runs seeded with and without GOOGLE have the advantage of having a lower mean rank and variance than GOOGLE (1.12 ± 0.15 and 1.14 ± 0.15, respectively, as opposed to GOOGLE's 2.00 ± 16.64). I attribute this difference to WAWA-IR's learning ability, which is able to bump home pages to the top of the list. Finally, this set of experiments shows how WAWA-IR can

[8]There is a car dealership with the URL `www.chuckanderson.com` that gets a very high score.

be used to personalize search engines by reorganizing the results they return as well as searching for nearby pages that score high.

Table 14 compares the performances of WAWA-IR's home-page finder under all the combinations of these two different settings: (i) a change in the Q-function used during reinforcement learning, and (ii) a change in the target score of pages that have the same host as the target home pages. The search engines used in these experiments are ALTAVISTA, EXCITE, INFOSEEK, LYCOS, and TEOMA. All the experiments reported in Table 14 use the 76 home-page finding advice rules described earlier in this chapter (See Appendix B for a complete listing of the rules). Table 13 defines the notations used in Table 14.

Table 13: Notation for the 2001 Experiments Reported in Table 14

RL-WAWA-Q	Reinforcement learning with WAWA-IR's Q-function
RL-STD-Q	Reinforcement learning with the standard Q-function
SL-3	Supervised learning with target host score = 3.0
SL-7	Supervised learning with target host score = 7.0

Table 14: Home-Page Finder Performances in 2001 under Different WAWA-IR Settings. See Table 13 for Definitions of the terms in the "Setting" column.

WAWA-IR **Setting**	**% Found**	**Mean Rank Given Page Found**	**Avg Pages Fetched Before Home Page**
RL-WAWA-Q *SL-3*	91%	1.14 ± 0.15	22
RL-STD-Q *SL-3*	89%	1.19 ± 0.30	28
RL-WAWA-Q *SL-7*	88%	1.40 ± 0.68	23
RL-STD-Q *SL-7*	86%	1.48 ± 0.99	29

WAWA-IR performs better with its own beam-search Q-function than with the standard hill-climbing Q-function of reinforcement learning. For example, when employing its own Q-function (and setting its target host score to 3), WAWA-IR only needs to fetch an average of 22 pages before it finds the target home pages, as opposed to needing an average of 28 pages with the standard Q-function.

The final set of experiments shows the sensitivity of WAWA-IR's performance to my chosen supervised-learning heuristic. With all other settings fixed, WAWA-IR experiments with target host score of 3 find more home pages than those with target host score of 7. This result shows that for the SCOREPAGE network, a target host page is just another page that is *not* the target page and should not be given such a high score.

5.5 Summary

These experiments illustrate how the generic WAWA-IR system can be used to rapidly create an effective "home-page finder" agent. I believe that many other useful specialized IR agents can be easily created simply by providing task-specific advice to the generic WAWA-IR system. In particular, the GOOGLE experiment shows how WAWA-IR can be used as a post-processor to Web search engines by learning to reorganize the search engines' results (for a specific task) and searching for nearby pages that score high. In this manner, WAWA-IR can be trained to become a personalized search engine.

One cost of using my approach is that I fetch and analyze many Web pages. I have not focused on speed in my experiments, ignoring such questions as how well can WAWA-IR's homepage finder perform when it only fetches the capsule summaries that search engines return, etc.

Chapter 6

Using WAWA to Extract Information from Text

Information extraction (IE) is the process of pulling desired pieces of information out of a document, such as the name of a disease or the location of a seminar (Lehnert 2000). Unfortunately, building an IE system requires either a large number of annotated examples[1] or an expert to provide sufficient and correct knowledge about the domain of interest. Both of these requirements make it time-consuming and difficult to build an IE system.

Similar to the IR case, I use Wawa's theory-refinement mechanism to build an IE system, namely Wawa-IE.[2] By using theory refinement, I am able to strike an effective balance between needing a large number of labeled examples and having a complete and correct set of domain knowledge.

Wawa-IE takes advantage of the intuition that specialized IR problems are nearly inverse of IE problems. The general IR task is nearly an inverse of the *keyword/keyphrase extraction* task, where the user in interested in a set of descriptive words or phrases describing a document. I illustrate this intuition with an example. Assume we have access to an accurate home-page finder, which takes as input a person's name and returns her home page. The inverse of such an IR system is an IE system that takes in home pages and returns the names of the people to whom the pages belong. By using a *generate-and-test*

[1] By annotated examples, I mean the result of the tedious process of reading the training documents and tagging each extraction by hand.

[2] Portions of this chapter were previously published in Eliassi-Rad and Shavlik (2001a, 2001b).

approach to information extraction, I am able to utilize what is essentially an IR system to address the IE task. In the *generation* step, the user first specifies the slots to be filled (along with their part-of-speech tags or parse structures), then Wawa-IE generates a list of candidate extractions from the document. Each entry in this list of candidate extractions is one complete set of slot fillers for the user-defined extraction template. In the *test* step, Wawa-IE scores each possible entry in the list of candidate extractions *as if they were keyphrases given to an IR system*. The candidates that produce scores that are greater than a Wawa-learned threshold are returned as the extracted information.

Building an IR agent for the IE task is straightforward in Wawa. The user provides a set of advice rules to Wawa-IE, which describes how the system should score possible bindings to the slots being filled during the IE process. I will call the names of the slots to be filled *variables*, and use "binding a variable" as a synonym for "filling a slot." These initial advice rules are then "compiled" into the ScorePage network, which rates the goodness of a document in the context of the given variable bindings. Recall that ScorePage is a supervised learner. It learns by being trained on user-provided instructions and user-labeled pages. The ScoreLink network is not used in Wawa-IE since the IE task is only concerned with extracting pieces of text from documents.

Like its Wawa-IR agents, Wawa-IE agents do not blindly follow user's advice, but instead the agents refine the advice based on the training examples. The use of user-provided advice typically leads to higher accuracy from fewer user-provided training examples (see Chapter 7).

As already mentioned in Section 3.2.2, of particular relevance to my approach is the fact that Wawa-IE's advice language contains *variables*. To understand how Wawa-IE uses variables, assume that I want to extract speaker names from a collection of seminar announcements. I might wish to give such a system some advice like:

```
when (consecutive( Speaker · ?Speaker) AND
      nounPhrase(?Speaker)) then show page
```

The leading question marks indicate the slot to be filled, and '·' matches any single word. Also, recall that the advice language allows the user to specify the required part of speech tag or parse structure for a slot. For example, the predicate `nounPhrase(`*?Speaker*`)` is true only if the value bound to *?Speaker* is a noun phrase. The condition of my example rule matches phrases like "*Speaker is Joe Smith*" or "*Speaker*: *is Jane Doe*".[3]

Figure 14 illustrates an example of extracting speaker names from a seminar announcement using WAWA-IE. The announcement is fed to the candidate generator and selector, which produces a list of speaker candidates. Each entry in the candidates list is then bound to the variable *?Speaker* in advice. The output of the (trained) network is a real number (in the interval of -10.0 to 10.0) that represents WAWA-IE's confidence in the speaker candidate being a correct slot filler for the given document.

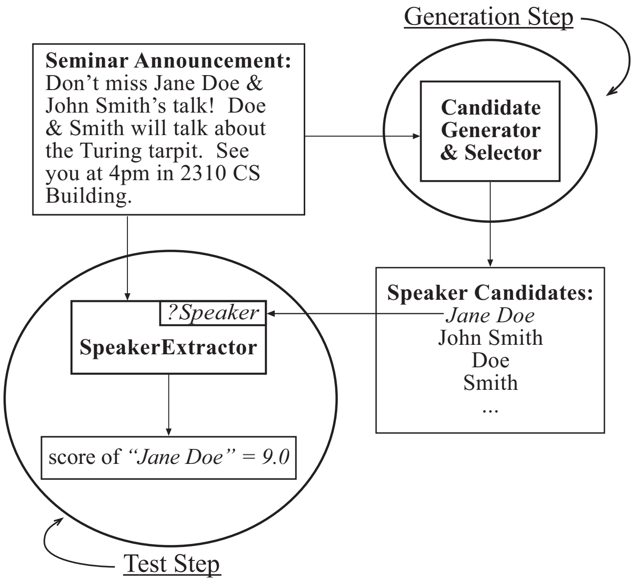

Figure 14: Extraction of Speaker Names with WAWA-IE

[3]WAWA's document-parser treats punctuation characters as individual tokens.

6.1 IE System Description

Wawa-IE uses a candidate generator and selector algorithm along with the ScorePage network to build IE agents. Table 15 provides a high-level description of Wawa-IE.

Table 15: Wawa's Information-Extraction Algorithm

1. Compile user's initial advice into the ScorePage network. 2. Run the *candidate generator and selector* on the training set and by using the untrainted ScorePage network from step 1 find negative training examples. 3. Train the ScorePage network on the user-provided positive training examples and the negative training examples generated in step 2. 4. Use a tuning set to learn the threshold on the output of the ScorePage network. 5. Run the *candidate generator and selector* on the test set to find extraction candidates for the test documents. 6. Using the trained ScorePage network (from step 3), score each test-set extraction candidate (produced in step 4). 7. Report the test-set extraction candidates that score above a system-learned threshold to the user.

To build and test an IE agent, Wawa-IE requires the user to provide the following information:

- The set of on-line documents from which the information is to be extracted.
- The extraction slots like speaker names, etc.

- The possible part-of-speech (POS) tags (*e.g.*, noun, proper noun, verb, etc) or parse structures (*e.g.*, noun phrase, verb phrase, etc) for each extraction slot.

- A set of advice rules which refer to the extraction slots as variables.

- A set of annotated examples, *i.e.*, training documents in which extraction slots have been marked.

Actually, the user does not have to explicitly provide the extraction slots and their POS tags separately from advice since they can be automatically extracted from the advice rules.

During training, Wawa-IE first compiles the user's advice into the ScorePage network. Wawa-IE next uses what I call an *individual-slot candidate generator* and a *combination-slots candidate selector* to create training examples for the ScorePage network. The individual-slot candidate generator produces a list of candidate extractions for each slot in the IE task. The combination-slots candidate selector picks candidates from each list produced by the individual-slot candidate generator and combines them to produce a single list of candidate extractions for the all the slots in the IE tasks. The same candidate generation and selection process is used after training to generate the possible extractions that the trained network[4]

During testing, given a document from which I wish to extract information, I generate a large number of candidate bindings, and then in turn I provide each set of bindings to the trained network. The neural network produces a numeric output for each set of bindings. Finally, my extraction process returns the bindings that are greater than a system-learned threshold.

[4]I use the terms "trained network" and "trained agent" interchangeably throughout Sections 6 and 7, since the network represents the agent's knowledge-base.

6.2 Candidate Generation

The first step WAWA-IE takes (both during training and after) is to generate all possible *individual* fillers for each slot on a given document. These candidate fillers can be individual words or phrases. Recall that in WAWA-IE, an extraction slot is represented by user-provided variables in the initial advice. Moreover, the user can provide syntactic information (*e.g.*, part-of-speech tags) about the variables representing extraction slots. WAWA-IE uses a slot's syntactic information along with either a part-of-speech (POS) tagger (Brill 1994) or a sentence analyzer (Riloff 1998) to collect the slot's candidate fillers.

For cases where the user specified POS tags[5] for a slot (*i.e.*, noun, proper noun, verb, etc), I first annotate each word in a document with its POS using Brill's tagger (1994). Then, for each slot, I collect every word in the document that has the same POS tag as the tag assigned to this variable at least once somewhere in the IE task's advice.

If the user indicated a parse structure for a slot (*i.e.*, noun phrase, verb phrase, etc), then I use *Sundance* (Riloff 1998), which builds a shallow parse tree by segmenting sentences into noun, verb, or prepositional phrases. I then collect those phrases that match the parse structure for the extraction slot and also generate all possible subphrases of consecutive words (since Sundance only does shallow parsing).

The user can provide both parse structures and POS tags for an extraction slot. In these cases, I run both Brill's tagger and Sundance sentence analyze (as described above) to get extraction candidates for the slot. I then merge and remove the duplicates in the lists provided by the two programs. In addition, the user can provide POS tags for some of the extraction slots and parse structures for others.

[5]The POS tags provided by the user for an extraction slot can be any POS tag defined in Brill's tagger.

For example, suppose the user specifies that the extraction slot should contain either a word tagged as a proper noun or two consecutive words both tagged as proper nouns. After using the Brill's tagger on the user-provided document, I then collect all the words that were tagged as proper nouns, in addition to every sequence of two words that were both tagged as proper nouns. So, if the phrase "Jane Doe" appeared on the document and the tagger marked both words as proper nouns, I would collect "Jane," "Doe," and "Jane Doe."

6.3 Candidate Selection

After the candidate generation step, WAWA-IE typically has lengthy lists of candidate fillers for each slot, and needs to focus on selecting good *combinations* that fill *all* the slots. Obviously, this process can be combinatorially demanding especially during training of a WAWA-IE agent, where backpropagation learning occurs multiple times over the entire training set. To reduce this computational complexity, WAWA-IE contains several methods (called *selectors*) for creating complete assignments to the slots from the lists of individual slot bindings.

WAWA-IE's selectors range from suboptimal and cheap (like simple random sampling from each individual list) to optimal and expensive (like exhaustively producing all possible combinations of the individual slot fillers). Among its heuristically inclined selectors, WAWA-IE has: (*i*) a modified WalkSAT algorithm (Selman, Kautz, and Cohen 1996), (*ii*) a modified GSAT algorithm (Selman, Kautz, and Cohen 1996), (*iii*) a hill-climbing algorithm with random restarts (Russell and Norvig 1995), (*iv*) a stochastic selector, and (*v*) a high-scoring simple-random-sampling selector. Section 7 provides a detailed discussion of the advantages and disadvantages of each selector within the context of my case studies.

I included WalkSAT and GSAT algorithms into my set of selectors for two reasons. First, the task of selecting combination-slots candidates is analogous to the problem of finding assignment for *conjunctive normal form* (CNF) formulas.

When selecting combination-slots candidates, I am looking for assignments that produce the highest score on the ScorePage network. Second, both WalkSAT and GSAT algorithms have been shown to be quite effective in finding assignment for certain classes of CNF formulas (Selman, Kautz, and Cohen 1996). The hill-climbing algorithm with random restarts was included into Wawa-IE's set of selectors because it has been shown to be an effective search algorithm (Russell and Norvig 1995). The stochastic selector was included because it utilizes the statistical distribution of extraction candidates as it pertains to their scores. The high-scoring simple-random-sampling selector was added since it is a very simple heuristic-search algorithm.

Figure 15 describes my *modified WalkSAT* algorithm. Wawa-IE builds the list of combination-slots candidate extractions for a document by randomly selecting an item from each extraction slot's list of individual-slot candidates. This produces a combination-slots candidate extraction that contains a candidate filler for each slot in the template. If the score produced by the ScorePage network is high enough (i.e., over a user-provided threshold) for this set of variable bindings, then Wawa-IE adds this combination to the list of combination-slots candidates. Otherwise, it repeatedly and randomly selects a slot in the template. Then, with probability p, Wawa-IE randomly selects a candidate for the selected slot and adds the resulting combination-slots candidate to the list of combination-slots candidates. With probability 1-p, it iterates over all possible candidates for this slot and adds the candidate that produces the highest network score for the document to the list of combination-slots candidates.

Figure 16 describes my *modified GSAT* algorithm. This algorithm is quite similar to my modified WalkSAT algorithm. In fact, it is the WalkSAT algorithm with $p = 0$. That is, if the score of the randomly-selected combination-slots candidate is not high enough, this algorithm randomly picks a slot and tries to find a candidate for the picked slot that produces the highest network score for the document.

I make two modifications to the standard WalkSAT and GSAT algorithms.

Inputs: MAX-TRIES, MAX-ALTERATIONS, *p*, MAX-CANDS, *doc*, *threshold*, *L* (where *L* is the lists of individual-slot candidate extractions for *doc*)

Output: *TL* (where *TL* is the list of combination-slots candidate extractions of size MAX-CANDS)

Algorithm:

1. *TL* := { }
2. **for** i:=1 **to** MAX-TRIES
    S := randomly selected combination-slots candidate from *L*.
    if (score of *S* w.r.t. *doc* is in [*threshold*, 10.0]), **then** add *S* to *TL*.
    otherwise
        for j:=1 **to** MAX-ALTERATIONS
            s := Randomly select a slot in *S* to change
            With probability *p*, randomly select a candidate for *s*.
                Add *S* to *TL*.
            With probability 1-*p*, select the first candidate for *s* that
                maximizes the score of *S* w.r.t. *doc*. Add *S* to *TL*.
3. **Sort** *TL* in decreasing order of score of its entries.
4. **Return** the top MAX-CANDS entries as *TL*.

Figure 15: Wawa-IE's Modified WalkSAT Algorithm

First, the default WalkSAT and GSAT algorithms check to see if an assignment was found that satisfies the CNF formula. In my version, I check to see if the score of the ScorePage network is above a user-provided threshold.[6] Second, the default WalkSAT and GSAT algorithms return after finding one assignment that satisfies the CNF formula. In my version, I collect a list of candidates and return the top-scoring N candidates (where N is defined by the user).

Figure 17 shows Wawa-IE's *hill-climbing algorithm with random restarts*. In this selector, Wawa-IE randomly selects a set of values for the combination-slots extraction candidate. Then, it tries to "climb" towards the candidates

[6]In my experiments, I used a threshold of 9.0.

Inputs: MAX-TRIES, MAX-ALTERATIONS, *p*, MAX-CANDS, *doc*, *threshold*, *L* (where *L* is the lists of individual-slot candidate extractions for *doc*)

Output: *TL* (where *TL* is the list of combination-slots candidate extractions of size MAX-CANDS)

Algorithm:

1. *TL* := { }
2. **for** i:=1 **to** MAX-TRIES
   S := randomly selected combination-slots candidate from *L*.
   if (score of *S* w.r.t. *doc* is in [*threshold*, 10.0]), **then** add *S* to *TL*.
   otherwise
      for j:=1 **to** MAX-ALTERATIONS
         s := Randomly select a slot in *S* to change
         Select the first candidate for *s* that maximizes the
            score of *S* w.r.t. *doc*. Add *S* to *TL*.
3. **Sort** *TL* in decreasing order of score of its entries.
4. **Return** the top MAX-CANDS entries as *TL*.

Figure 16: Wawa-IE's Modified GSAT Algorithm

that produce the high scores by comparing different assignments for each extraction slot.[7] When Wawa-IE cannot "climb" any higher or has "climbed" `MAX-CLIMBS` times, it restarts from another randomly chosen point in the space of combination-slots extraction candidates.

The basic idea behind the next selector, namely *stochastic selector*, is to estimate the *goodness* (with respect to the output of the ScorePage network) of a candidate for a *single* slot by averaging over multiple random candidate bindings for the other slots in the extraction template. For example, what is the expected score of the protein candidate "LOS1" when the location candidate is randomly selected? Figure 18 describes Wawa-IE's *stochastic selector*. For each slot in the extraction template, Wawa-IE first uniformly samples from

[7]In this selector and the stochastic selector (described next), the *score*(S) function refers to the output of the ScorePage network for the combination-slots extraction candidate, S.

Inputs: MAX-TRIES, MAX-CLIMBS, MAX-CANDS, *doc*, *L* (where *L* is the lists of individual-slot candidate extractions for *doc*)

Output: *TL* (where *TL* is the list of combination-slots candidate extractions of size MAX-CANDS)

Algorithm:

1. *TL* := { }
2. **for** i:=1 **to** MAX-TRIES
    S := randomly selected combination-slots candidate from *L.*
    best_S = *S.*
    max_score := score of *S* w.r.t. *doc*.
    prev_score := *max_score.*
    for j:=1 **to** MAX-CLIMBS
      for all slots *s* in *S*
        max_cand(*s*) := the first candidate for *s* that maximizes the score of *S* w.r.t. *doc*.
        S' := *S* with the candidate *max_cand*(*s*) filling slot *s.*
        max_score := **max**(*max_score*, score of *S'*) .
        if (*max_score* == score of *S'*) **then** *best_S* = *S'*.
      if (*max_score* == *prev_score*),
        then add *S* to *TL* and break out of the inner loop.
      otherwise
        S := *best_S*.
        prev_score := *max_score.*
3. **Sort** *TL* in decreasing order of score of its entries.
4. **Return** the top MAX-CANDS entries as *TL*.

Figure 17: Wawa-IE's Hill-Climbing Algorithm With Random Restarts

the list of individual-slot candidates. The uniform selection allows Wawa-IE to accurately select an initial list of sampled candidates. That is, if a candidate occurs more than once on the page, then it should have a higher probability of getting picked for the sampled list of candidates. The size of this sample is determined by the user.

Then, Wawa-IE attempts to estimate the probability of picking a candidate for each individual-slot candidate list and iteratively defines it to be

$$P_k(c_i \text{ will be picked}) = \frac{\overline{score_k(c_i)}}{\sum_{j=1}^{N} \overline{score_k(c_j)}} \qquad (6.1)$$

where $\overline{score_k(c_i)}$ is the *mean score of candidate* c_i at the k^{th} try and is defined as

$$\overline{score_k(c_i)} = \frac{(\sum_{m=1}^{M} score_m(c_i)) + (\overline{score_{prior}} \times m_{prior})}{n + m_{prior}}$$

The function $score(c_i)$ is equal to the output of the ScorePage network, when candidate i is assigned to slot s_e and values for all the other slots are picked based on their probabilities.[8] M is the number of times candidate i was picked. N is the number of unique candidates in the initial sampled set for slot s_e and n is the total number of candidates in the initial sampled set for slot s_e. $\overline{score_{prior}}$ is an estimate for the *prior mean score* and m_{prior} is the *equivalent sample size* (Mitchell 1997).[9] The input parameters `MAX-SAMPLE` and `MAX-TIME-STEPS` are defined by the user. They, respectively, determine the size of the list sampled initially from an *individual* slot's list of candidates and the number of times Equation 6.1 should be updated. Note that `MAX-SAMPLE` and `MAX-TIME-STEPS` are analogous to `MAX-TRIES` and `MAX-ALTERATIONS` in Wawa-IE's modified walkSAT and GSAT algorithms.

Wawa-IE's *high-scoring simple-random-sampling* selector randomly picks N combination-slots candidates from the lists of individual-slot candidates, where N is provided by the user. When training, it only uses the N combinations that produce the highest scores on the *untrained* ScorePage network.[10]

Wawa-IE does not need to generate combinations of fillers when the IE task contains a template with only one slot (as is the case in one of my case studies presented in Section 7). However, it is desirable to trim the list of candidate fillers during the training process because training is done iteratively. Therefore, Wawa-IE heuristically selects from a slot's list of *training* candidate

[8]Score of a candidate is mapped into $[0, 1]$.

[9]In my experiments, $\overline{score_{prior}} = 0.75$ and $m_{prior} = 10$ for words with prior knowledge (*i.e.*, words that are labeled by the user as possibly relevant). For all other words, $\overline{score_{prior}} = 0.5$ and $m_{prior} = 3$.

[10]By untrained, I mean a network containing only compiled (initial) advice and without any further training via backpropagation and labeled examples.

Inputs: MAX-SAMPLE, MAX-TIME-STEPS, MAX-CANDS, *doc*, *L* (where *L* is the lists of individual-slot candidate extractions for *doc*)

Output: *TL* (where *TL* is the list of combination-slots candidate extractions of size MAX-CANDS)

Algorithm:

1. *TL* := { }
2. **for each** slot s_e in the list of all slots provided by the user

    $Sampled\_Set_e$:= { }

    for i:=1 **to** MAX-SAMPLE

        Append to $Sampled\_Set_e$ a uniformly selected candidate from s_e.

    for k:=1 **to** MAX-TIME-STEPS

        for each candidate c_i in $Sampled\_Set_e$

            Calculate the probability of picking c_i at the k^{th} attempt using Equation 6.1

3. **for** j = 1 **to** MAX-CANDS

    S := combination-slots candidate stochastically chosen according to Equation 6.1

    Add *S* to *TL*.

4. **Return** the top MAX-CANDS entries as *TL*.

Figure 18: WAWA-IE's Stochastic Selector

fillers (*i.e.*, the candidate fillers associated with the training set) by scoring each candidate filler using the untrained SCOREPAGE network[11] and returning the highest scoring candidates plus some randomly sampled candidates. This process of picking *informative* candidate fillers from the training data has some beneficial side effects, which are described in more detail in the next section.

[11]By untrained, I mean a network containing only compiled (initial) advice and without any further training via backpropagation and labeled examples.

6.4 Training an IE Agent

Figure 19 shows the process of building a trained IE agent. Since (usually) only positive training examples are provided in IE domains, I first need to generate some negative training examples.[12] To this end, I use the candidate generator and selector described above. The user selects which selector she wants to use during training. The list of negative training examples collected by the user-picked selector contains informative negative examples (*i.e.*, near misses) because the heuristic search used in the selector scores the training documents on the untrained ScorePage network. That is, the (user-provided) prior knowledge scored these "near miss" extractions highly (as if they were true extractions).

After the N highest-scoring negative examples are collected, I train the ScorePage neural network using these negative examples and all the provided positive examples. By training the network to recognize (i.e., produce a high output score for) a correct extraction in the context of the document as a whole (see Section 3.4), I am able to take advantage of the global layout of the information available in the documents of interest.

Since the ScorePage network outputs a real number, Wawa-IE needs to learn a threshold on this output such that the bindings for the scores above the threshold are returned to the user as extractions and the rest are discarded. Note that the value of the threshold can be used to manipulate the performance of the IE agent. For example, if the threshold is set to a high number (*e.g.*, 8.5), then the agent might miss a lot of the correct fillers for a slot (*i.e.*, have low *recall*), but the number of extracted fillers that are correct should be higher (*i.e.*, high *precision*). As previously defined in Chapter 2, *Recall* (van Rijsbergen 1979) is the ratio of the number of correct fillers extracted to the total number of fillers in correct extraction slots. *Precision* (van Rijsbergen 1979) is the ratio

[12] Wawa-IE needs negative training examples because it frames the IE task as a classification problem.

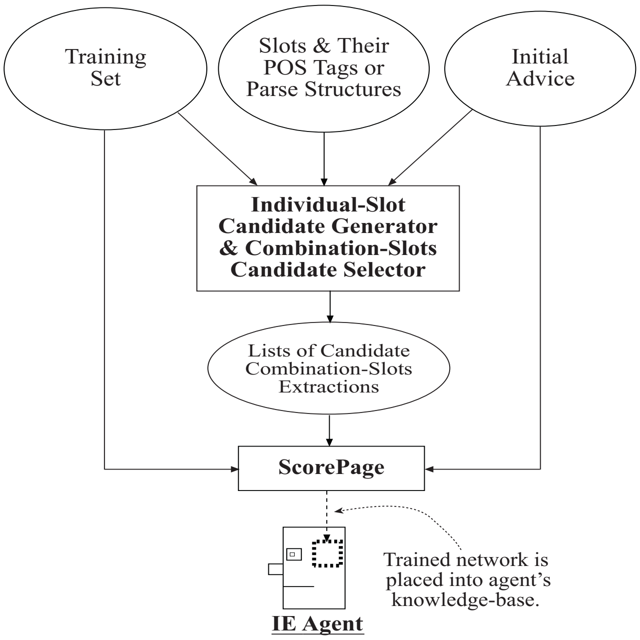

Figure 19: Building a Trained IE agent

of the number of correct fillers extracted to the total number of fillers extracted.

To avoid overfitting the ScorePage network and to find the best threshold on its output after training is done, I actually divide the training set into two disjoint sets. One of the sets is used to train the ScorePage network. The other set, the *tuning* set, is first used to "stop" the training of the ScorePage network. Specifically, I cycle through the training examples 100 times. After each iteration over the training examples, I use the lists of candidate fillers associated with the tuning set to evaluate the F_1-*measure* produced by the network for various settings of the threshold. Recall that, the F_1-*measure* combines precision and recall using the following formula: $F_1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$.[13] I pick the network that produced the highest F_1-measure on my tuning set as my final trained network.

[13]The F_1-measure (van Rijsbergen 1979) is used regularly to compare the performances of IR and IE systems because it weights precision and recall equally and produces one single number.

I utilize the tuning set (a second time) to find the optimal threshold on the output of the trained ScorePage network. Specifically, I perform the following:

- For each *threshold* value, t, from -10.0 to 10.0 with increments of *inc*, do
  - Run the tuning set through the trained ScorePage network to find the F_1-measure (for the threshold t).
- Set the optimal threshold to the threshold associated with the maximum F_1-measure.

The value of the increment, *inc*, is also defined during tuning. The initial value of *inc* is 0.25. Then, if the variation among the F_1-measures calculated on the tuning set is small (*i.e.*, less than 0.05 and F_1-measure $\in$ [0,1]), I reduce the *inc* by 0.05. Note that, the smaller *inc* is, the more accurately the trained ScorePage can be evaluated (because the more sensitive it is to the score of a candidate extraction).

6.5 Testing a Trained IE Agent

Figure 20 depicts the steps a trained IE agent takes to produce extractions. For each entry in the list of combination-slots extraction candidates, Wawa-IE first binds the variables to their candidate values. Then, it performs a forward propagation on the trained ScorePage network and outputs the score of the network for the test document based on the candidate bindings. If the output value of the network is greater than the threshold defined during the tuning step, Wawa-IE records the bindings as an extraction. Otherwise, these bindings are discarded.

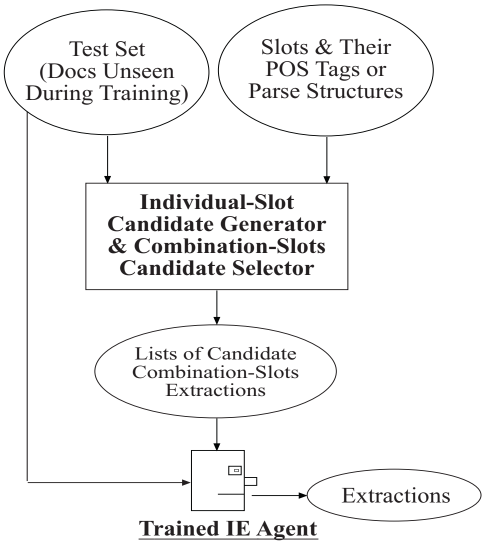

Figure 20: Testing a Trained IE Agent

6.6 Summary

A novel aspect of WAWA-IE is its exploitation of the relationship between IR and IE. That is, I build IR agents that treat possible extractions as keywords, which are in turn judged within the context of the entire document.

The use of theory refinement allows me to take advantage of user's prior knowledge, which need not be perfectly correct since WAWA-IE is a learning system. This, in turn, reduces the need for labeled examples, which are very expensive to get in the IE task. Also, compiling users' prior knowledge into the SCOREPAGE network provides a good method for finding informative negative training examples (*i.e.*, near misses).

One cost of using my approach is that I require the user to provide the POS tags or parse structures of the extraction slots. I currently assume that Brill's tagger and Sundance are perfect (*i.e.*, they tag words and parse sentences with 100% accuracy). Brill's tagger annotates the words on a document with 97.2% accuracy (Brill 1994), so 2.8% error rate propagates into my results. I was not able to find accuracy estimates for Sundance.

My approach is computationally demanding, due to its use of a generate-and-test approach. But, CPU cycles are abundant, and our experiments presented in Chapter 7 show that WAWA-IE still performs well when only using a subset of all possible combinations of slot fillers.

Chapter 7

Extraction Experiments with WAWA

This chapter[1] presents three case studies involving WAWA-IE. The first information-extraction (IE) task involves extracting speaker and location names from a collection of seminar announcements (Freitag 1998b). This task has been widely used in the literature and allows me to directly compare the performance of WAWA-IE to several existing systems. For this domain, I follow existing methodology and independently extract speaker and location names, because each document is assumed to contain only one announcement. That is, for each announcement, I do not try to pair up speakers and locations, instead I return a list of speakers and a separate list of locations. Hence, I do not use any "combination-slots selector" on this task.

The second and third IE tasks involve extracting information from abstracts of biomedical articles (Ray and Craven 2001). In the second IE task, protein names and their locations within the cell are to be extracted from a collection of abstracts on yeast. In the third IE task, genes names and the diseases associated with them are to be extracted from a collection of abstracts on human disorders. I chose these two domains because they illustrate a harder IE task than the first domain. In these domains, the fillers for extraction slots depend on each other because a single abstract can contain multiple fillers for a slot. Hence for each document, a single *list* of *<protein*, *location> pairs* is extracted for the second

[1]Portions of this chapter were previously published in Eliassi-Rad and Shavlik (2001a, 2001b).

domain and a single *list* of <*gene*, *disease*> *pairs* is extracted for the third domain.

7.1 CMU Seminar-Announcement Domain

This section describes experimental results that I performed to test WAWA-IE on the CMU seminar-announcements domain (Freitag 1998b). This domain consists of 485 documents. The complete task is to extract the start time, end time, speaker, and location from an announcement. However, I only report on results for extracting speaker and location. I omit start and end times from my experiments since almost all systems perform very well on these slots.

Seminar announcements are tagged using Brill's part-of-speech tagger (1994) and common ("stop") words are discarded (Belew 2000). I did not stem the words in this study since converting words to their base forms removes information that would be useful in the extraction process. For example, I do not want the word "Professors" to stem to the word "Professor", since I want to give the IE agent the opportunity to learn that usually more than one speaker name appears after the word "Professors."

Experimental Methodology

I compare WAWA-IE to seven other information extraction systems using the CMU seminar announcements domain (Freitag 1998b). These systems are HMM (Freitag and McCallum 1999), BWI (Freitag and Kushmerick 2000), SRV (Freitag 1998b), Naive Bayes (Freitag 1998b), WHISK (Soderland 1999), RAPIER (Califf 1998), and RAPIER-WT (Califf 1998). None of these systems exploits prior knowledge. Except for Naive Bayes, HMM, and BWI, the rest of the systems use relational learning algorithms (Muggleton 1995). RAPIER-WT is a variant of RAPIER where information about semantic classes is not utilized. HMM (Freitag and McCallum 1999) employs a hidden Markov model to learn about extraction slots. BWI (Freitag and Kushmerick 2000) combines wrapper induction techniques (Kushmerick 2000) with AdaBoost (Schapire and Singer 1998) to solve the IE task.

Freitag (1998b) first randomly divided the 485 documents in the seminar announcements domain into ten splits, and then randomly divided each of the ten splits into approximately 240 training examples and 240 testing examples. Except for WHISK, the results of the other systems are all based on the same 10 data splits. The results for WHISK are from a single trial with 285 documents in the training set and 200 documents in the testing set.

I give WAWA-IE nine and ten advice rules in *Backus-Naur Form*, BNF, (Aho, Sethi, and Ullman 1986) notation about speakers and locations, respectively (see Appendix C). I wrote none of these advice rules with the specifics of the CMU seminar announcements in mind. The rules describe my prior knowledge about what might be a speaker or a location in a *general* seminar announcement. It took me about half a day to write these rules and I did not *manually* refine these rules over time.[2].

For this case study, I choose to create the same number of negative training examples (for speaker and location independently) as the number of positive examples. I choose 95% of the negatives, from the complete list of possibilities, by collecting those that score the highest on the untrained SCOREPAGE network; the remaining 5% are chosen randomly from the complete list.

For this domain, I used four variables to learn about speaker names and four variables to learn about location names. The four variables for speaker names refer to first names, nicknames, middle names (or initials), and last names, respectively. The four variables for location refer to a cardinal number (namely *?LocNumber*) and three other variables representing the non-numerical portions of a location phrase (namely, *?LocName1, ?LocName2, and ?LocName3*). For example, in the phrase "1210 West Dayton," *?LocNumber, ?LocName1, and*

[2]I also wrote a set of rules for this domain. The rules are lists in Section2 of Appendix D. The results are reported in Eliassi-Rad and Shavlik (2001b)

?LocName2 get bound to 1210, "West," and "Dayton" respectively.

Table 16 shows four rules used in the domain theories of speaker and location slots. Rule SR1 matches phrases of length three that start with the word "Professor" and have two proper nouns for the remaining words.[3] In rule SR2, I am looking for phrases of length four where the first word is "speaker," followed by another word which I do not care about, and trailed by two proper nouns. SR2 matches phrases like "*Speaker : Joe Smith*" or "*speaker is Jane Doe.*" Rules LR1 and LR2 match phrases such as "*Room 2310 CS.*" LR2 differs from LR1 in that it requires the two words following "room" to be a cardinal number and a proper noun, respectively (i.e., LR2 is a subset of LR1). Since I am more confident that phrases matching LR2 describe locations, LR2 sends a higher weight to the output unit of the ScorePage network than does LR1.

Table 16: Sample Rules Used in the Domain Theories of Speaker and Location Slots

SR1	When "Professor *?FirstName*/NNP *?LastName*/NNP" then strongly suggest showing page
SR2	When "Speaker . *?FirstName*/NNP *?LastName*/NNP" then strongly suggest showing page
LR1	When "Room *?LocNumber ?LocName*" then suggest showing page
LR2	When "Room *?LocNumber*/CD *?LocName*/NNP" then strongly suggest showing page

Tables 17 and 18 show the results of the trained Wawa-IE agent and the other seven systems for the speaker and location slots, respectively.[4] The results reported are the averaged precision, recall, and F_1 values across the ten splits. The precision, recall, and F_1-measure for a split are determined by the optimal threshold found for that split using the tuning set (see Section 6.4 for further

[3]When matching preconditions of rules, case of words does not matter. For example, both "Professor alan turing" and "Professor Alan Turning" will match rule SR1's precondition.

[4]Due to the lack of statistical information on the other methods, I cannot statistically measure the significance of the differences in the algorithms.

details). For all ten splits, the optimal thresholds on WAWA-IE's untrained agent are 5.0 for the speaker slot and 9.0 for the location slot. The optimal threshold on WAWA-IE's trained agent varies from one split to the next in both the speaker slot and the location slot. For the speaker slot, the optimal thresholds on WAWA-IE's trained agent vary from 0.25 to 2.25. For the location slot, the optimal thresholds on WAWA-IE's trained agent range from -6.25 to 0.75.

Since the speaker's name and the location of the seminar may appear in multiple forms in an announcement, an extraction is considered correct as long as any one of the possible correct forms is extracted. For example, if the speaker is "John Doe Smith", the words "Smith", "Joe Smith", "John Doe Smith", "J. Smith", and "J. D. Smith" might appear in a document. Any one of these extractions is considered correct. This method of marking correct extractions is also used in the other IE systems against which I compare my approach.

I use precision, recall, and the F_1-measure to compare the different systems (see Section 6.4 for definitions of these terms). Recall that an ideal system has precision and recall of 100%.

Table 17: Results on the Speaker Slot for Seminar Announcements Task

System	Precision	Recall	F_1
HMM	77.9	75.2	76.6
WAWA-IE's Trained Agent	61.5	86.6	71.8
BWI	79.1	59.2	67.7
SRV	54.4	58.4	56.3
RAPIER-WT	79.0	40.0	53.1
RAPIER	80.9	39.4	53.0
WAWA-IE's Untrained Agent	29.5	96.8	45.2
Naive Bayes	36.1	25.6	30.0
WHISK	71.0	15.0	24.8

The F_1-measure is more versatile than either precision or recall for explaining relative performance of different systems, since it takes into account the inherent

Table 18: Results on the Location Slot for Seminar Announcements Task

System	Precision	Recall	F_1
WAWA-IE's Trained Agent	73.9	84.4	78.8
HMM	83.0	74.6	78.6
BWI	85.4	69.6	76.7
RAPIER-WT	91.0	61.5	73.4
SRV	74.5	70.1	72.3
RAPIER	91.0	60.5	72.7
WHISK	93.0	59.0	72.2
RAPIER-W	90.0	54.8	68.1
Naive Bayes	59.6	58.8	59.2
WAWA-IE's Untrained Agent	29.2	98.2	45.0

tradeoff that exists between precision and recall. For both speaker and location slots, the F_1-measure of WAWA-IE is considerably higher than Naive Bayes and all the relational learners. WAWA-IE's trained agents perform competitively with the BWI and HMM learners. The F_1-measures on WAWA-IE's trained agents are high because I generate many extraction candidates in my generate-and-test model. Hence, the trained agents are able to extract a lot of the correct fillers from the data set, which in turn leads to higher recall than the other systems.

After training, WAWA-IE is able to reject enough candidates so that it obtains reasonable precision. Figure 21 illustrates this fact for the speaker slot. A point in this graph represents the averaged precision and recall values at a specific network output across the ten splits. There are 38 points on each curve in Figure 21 representing the network outputs from 0.0 to 9.0 with increments of 0.25. The trained agent generates much better precision scores than the untrained agent. This increase in performance from WAWA-IE's untrained agent to WAWA-IE's trained agent shows that my agent is not "hard-wired" to perform well on this domain and that training helped my performance.

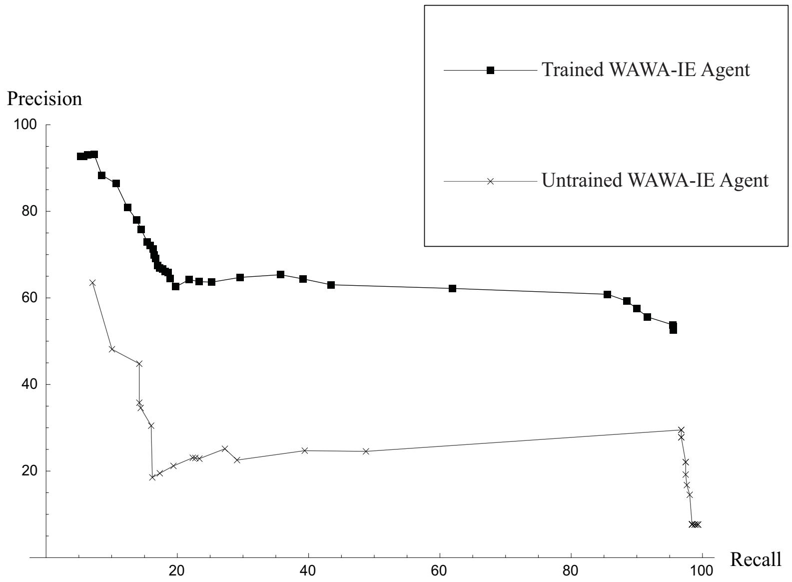

Figure 21: WAWA-IE's Speaker Extractor: Precision & Recall Curves

Finally, I should note that several of the other systems have higher precision, so depending on the user's tradeoff between recall and precision, one would prefer different systems on this testbed.

7.2 Biomedical Domains

This sections presents two experimental studies done on biomedical domains. Ray and Craven created both of these data sets (2001). In the first domain, the task is to extract protein names and their locations on the cell. In the second domain, the task is to extract genes and the genetic disorders with which they are associated.

Subcellular-Localization Domain

For my second experiment, the task is to extract protein names and their locations on the cell from Ray and Craven's *subcellular-localization* data set (2001).

Their extraction template is called the *subcellular-localization relation*. They created their data set by first collecting target instances of the subcellular-localization relation from the *Yeast Protein Database* (YPD) Web site. Then, they collected abstracts from articles in the *MEDLINE* database (NLM 2001) that have references to the entries selected from YPD.

In the subcellular-localization data set, each training and test instance is an individual sentence. A positive sentence is labeled with target tuples (where a tuple is an instance of the subcellular-localization relation). There are 545 positive sentences containing 645 tuples, of which 335 are unique. A negative sentence is not labeled with any tuples. There are 6,700 negative sentences in this data set. Note that a sentence that does not contain *both* a protein and its subcellular location is considered to be negative.

Wawa-IE is given 12 advice rules in BNF (Aho, Sethi, and Ullman 1986) notation about a protein and its subcellular location (see Appendix D for a complete list of rules). Michael Waddell, who is an MD/PhD student at the University of Wisconsin-Madison, wrote these advice rules for me. Moreover, I did not *manually* refine these rules over time.

Ray and Craven (2001) split the subcellular-localization data set into five disjoint sets and ran five-fold cross-validation. I use the same folds with Wawa-IE and compare my results to theirs.

Figure 22 compares the following systems as measured by precision and recall curves: (*i*) Wawa-IE's agent with no selector during training, (*ii*) Wawa-IE's agent with stochastic selector picking 50% of all possible negative combination-slots candidates during the training phase, and (*iv*) Ray and Craven's system (2001). In the Wawa-IE runs, I used all the positive training examples and 100% of all possible test-set combination-slots candidates.

The trained IE agent without any selector algorithm produces the best results. But, it is computationally expensive since it needs to take the cross-product of all entries in the lists of individual-slot candidates. The trained IE agents with the stochastic selector (picking 50% of the possible train-set negative combination-slots candidates) performs quite well, outperforming Ray and Craven's system.

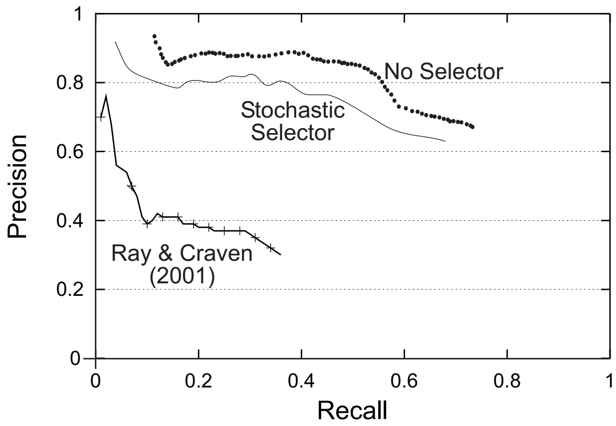

Figure 22: Subcellular-Localization Domain: Precision & Recall Curves for Ray and Craven's System (2001), WAWA-IE runs with no selector, with the stochastic selector sampling 50% of the possible negative training tuples.

OMIM Disorder-Association Domain

For my third experiment, the task is to extract gene names and their genetic disorders from Ray and Craven's *disorder-association* data set (2001). Their extraction template is called the *disorder-association relation.* They created their data set by first collecting target instances of the disorder-association relation from the *Online Mendelian Inheritance in Man* (OMIM) database (Center for Medical Genetics, 2001). Then, they collected abstracts from articles in the *MEDLINE* database (NLM 2001) that have references to the entries selected from OMIM.

In the disorder-association data set, each training and test instance is an individual sentence. A positive sentence is labeled with target tuples (where a tuple is an instance of the disorder-association relation). There are 892 positive sentences containing 899 tuples, of which 126 are unique. A negative sentence is not labeled with any tuples. There are 11,487 negative sentences in this data set. Note that a sentence that does not contain both a gene and its genetic disorder is considered to be negative.

WAWA-IE is given 14 advice rules in BNF (Aho, Sethi, and Ullman 1986) notation about a gene and its genetic disorder (see Appendix E for a complete list of rules). Michael Waddell, who is an MD/PhD student at the University of Wisconsin-Madison, wrote these advice rules for me. Moreover, I did not *manually* refine these rules over time.

Ray and Craven (2001) split the subcellular-localization data set into five disjoint sets and ran five-fold cross-validation. I use the same folds with WAWA-IE and compare my results to theirs.

Figure 23 compares WAWA-IE's agent with no selector and stochastic selector to that of Ray and Craven's (2001) as measured by precision and recall curves on the five test sets. In these runs, I used all the positive training examples and 50% of the negative training examples. The stochastic selector was used to pick the negative examples. The trained IE agent without any selector algorithm is competitive with Ray and Craven's system. WAWA-IE is able to out-perform Ray and Craven's system after 40% recall. The trained IE agent with stochastic selector (picking 50% of all possible negative train-set combination-slots candidates) performs competitively to Ray and Craven's at around 60% recall. In this domain, the trained IE agent with stochastic selector cannot reach the precision level achieved by Ray and Craven's system.

Ray and Craven (2001) observed that their system would get better recall if the sample number of negative sentences were used as the number of positive examples during training. Since I used their exact folds of training data in testing WAWA-IE, this means that the number of training sentences is approximately 1110 and 1800 in the yeast and OMIM domains, respectively.

The methodology used in the next two sections is slightly different than the the one described in this section. I made this modification to make WAWA-IE run faster. Basically, I consider a larger list of positive tuples than the one used

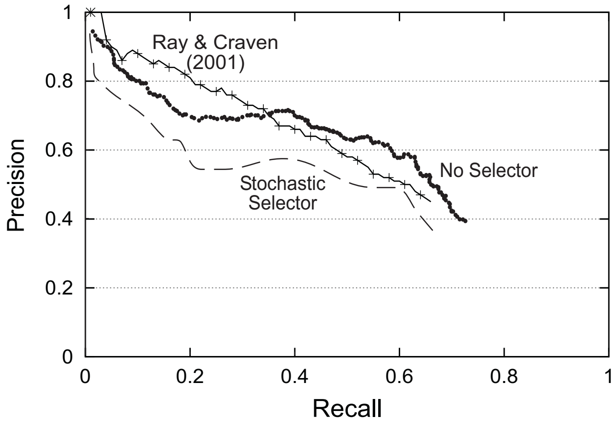

Figure 23: Disorder-Association Domain: Precision & Recall Curves for Ray and Craven's system (2001), WAWA-IE runs with no selector, with the stochastic selector sampling 50% of the possible negative training tuples.

in this section's experiments. For example, the tuples "⟨LOS1, nuclei⟩" and "⟨LOS1 protein, nuclei⟩" are thought of as one tuple in the experiments in this section and as two tuples in the experiments reported next.

7.3 Reducing the Computational Burden

This section reports on WAWA-IE's performance when all of the system-generated negative examples are not utilized. These experiments investigate whether WAWA-IE can intelligently select good negative training examples and hence reduce the computational burden (during the training process) by comparing test set F_1-measures to the case where all possible negative training examples are used.

Subcellular-Localization Domain

Figure 24 illustrates the difference in F_1-measure (from a five-fold cross-validation experiment) between choosing no selector and Section 6.3's stochastic, hill climbing, GSAT, WalkSAT, and uniform selectors. The horizontal axis depicts the percentage of negative training examples used during the learning process (100% of negative training examples is approximately 53,000 tuples), and the vertical axis depicts the F_1-measure of the trained IE-agent on the test set. In these runs, I used all of the positive training examples.

It is not a surprise that the trained IE agent without any selector algorithm produces the best results. But as mentioned before, it is computationally quite expensive, especially for tasks with many extraction slots. The best-performing selector was my stochastic selector, followed by the hill-climbing approach with multiple restarts. GSAT edged WalkSAT slightly in performance. The uniform selector has the worst performance. The stochastic selector out-performs all other selectors since it adaptively selects candidates based on the probability of how high they will score. As expected, WAWA-IE improves its F_1-measure on the test set (for all selectors) when given more negative training examples.

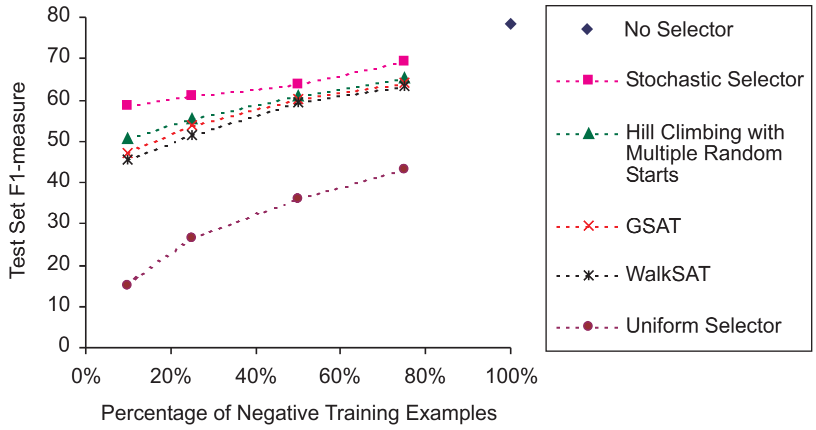

Figure 24: Subcellular-Localization Domain: F_1-measure versus Percentage of Negative Training Candidates Used for Different Selector Algorithms

OMIM Disorder-Association Domain

Figure 25 illustrates the difference in F_1-measure, again from a five-fold cross-validation experiment, between choosing no selector and Section 6.3's stochastic, hill climbing, GSAT, WalkSAT, and uniform selectors for the OMIM disorder-association domain. The horizontal axis depicts the percentage of negative training examples used during the learning process (100% of negative training examples is approximately 245,000 tuples), and the vertical axis depicts the F_1-measure of the trained IE-agent on the test set. In these runs, I used all of the positive training examples.

Again, the trained IE agent without any selector algorithm produces the best results. The best-performing selector was my stochastic selector, followed by the hill-climbing approach with multiple restarts, GSAT, WalkSAT, and the uniform selector. Again, the stochastic selector out-performs all other selector since it selects candidates based on the probability of how high they will score. Moreover, as expected, WAWA-IE improves its F_1-measure on the test set (for all selectors) when given more negative training examples.

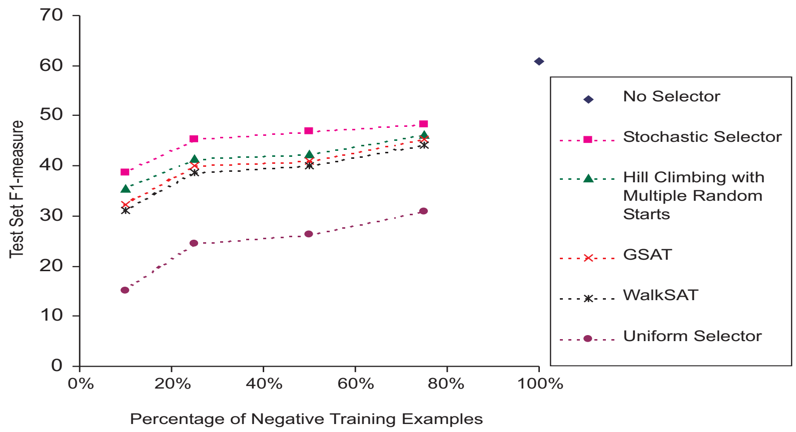

Figure 25: Disorder-Association Domain: F_1-measure versus Percentage of Negative Training Candidates Used for Different Selector Algorithms

7.4 Scaling Experiments

This section reports on WAWA-IE's performance when all of the positive training instances are not available or the user provides fewer advice rules.

Subcellular-Localization Domain

Figure 26 demonstrates WAWA-IE's ability to learn when positive training examples are sparse. The horizontal axis shows the number of positive training instances, and the vertical axis depicts the F_1-measure of the trained IE-agent averaged over the five test sets. The markers on the horizontal axis correspond to 10%, 25%, 50%, 75%, 100% of positive training instances, respectively. I measured WAWA-IE's performance on an agent with no selector. That is, all possible training and testing candidates were generated. WAWA-IE's performance degrades smoothly as the number of positive training examples decreases. This curve suggests that more training data will improve the quality of the results (since the curve is still rising).

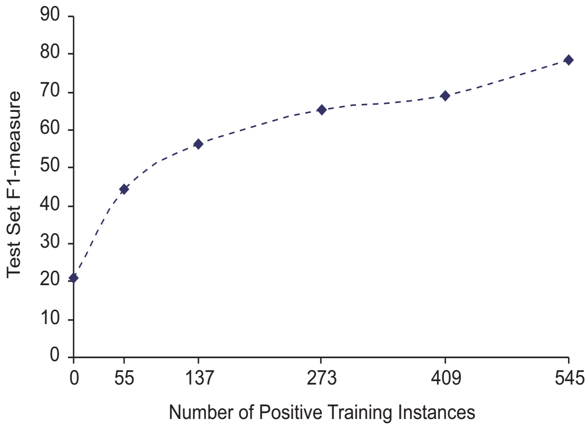

Figure 26: Subcellular-Localization Domain: F_1-measure versus Number of Positive Training Instances. The curve represents WAWA-IE's performance without a selector during training and testing.

Figure 27 shows how WAWA-IE performs on the test set F_1-measure with different types and number of advice rules. There are only two rules in group A. They include just mentioning the variables that represent the extraction slots in the template. There are five rules in group B. None of these rules refer to both the protein and location in one rule. In other words, there is no initial advice relating proteins and their locations. There are 12 rules in advice of type C. They include rules from groups A and B, plus seven more rules giving information about proteins and locations together (see Appendix D for the full set of rules used in this experiment). I used all the positive and system-generated negative training examples for this experiment. WAWA-IE with no selector during training and testing is able to learn quite well with very minimal advice, which shows that the advice rules do not have "hard-wired" in them the correct answers to the extraction task.

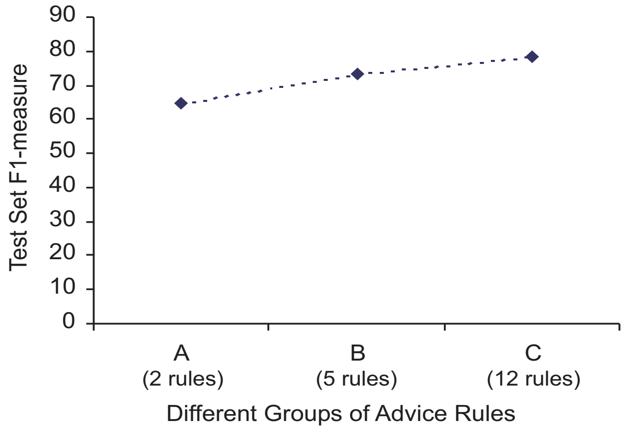

Figure 27: F_1-measure versus Different Groups of Advice Rules on WAWA-IE with no Selector. Groups A, B, and C have 2, 5, and 12 rules, respectively. The curve represnts WAWA-IE's performance without a selector during training and testing.

OMIM Disorder-Association Domain

Figure 28 demonstrates WAWA-IE's ability to learn when positive training examples are sparse. The horizontal axis shows the number of positive training instances, and the vertical axis depicts the F_1-measure of the trained IE-agent averaged over five test sets. The markers on the horizontal axis correspond to 10%, 25%, 50%, 75%, 100% of positive training instances, respectively. I measured WAWA-IE's performance on an agent with no selector during training and testing. WAWA-IE's performance without a selector degrades smoothly as the number of positive training examples decreases. Similar to the curve for the subcellular-localization domain, this curve suggests that more labeled data will improve the quality of results (since the curve is still steeply rising).

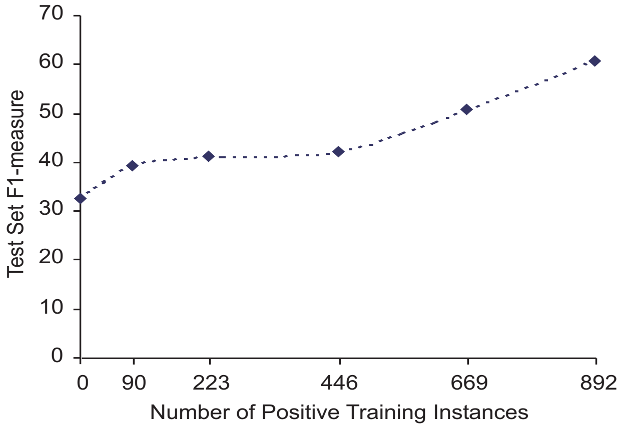

Figure 28: Disorder-Association Domain: F_1-measure versus Number of Positive Training Instances. The curve represents WAWA-IE's performance without a selector during training and testing.

Figure 29 shows how WAWA-IE performs on the test set F_1-measure, averaged across the five test sets, with different types and number of advice rules. There are only two rules in group A. They include just mentioning the variables that represent the extraction slots in the template. There are eight rules in group B. None of these rules refer to both genes and their genetic disorders in one rule. In other words, there is no initial advice relating genes and their genetic disorders. There are 14 rules in advice of type C. They include rules from groups A and B, plus six more rules giving information about genes and their disorders together (see Appendix E for the full set of rules used in this experiment). I used all the positive and system-generated negative training examples for this experiment. As was the case in the subcellular-localization domain, WAWA-IE is able to learn quite well with very minimal advice, which shows that the advice rules do not have "hard-wired" in them the correct answers to the extraction task.

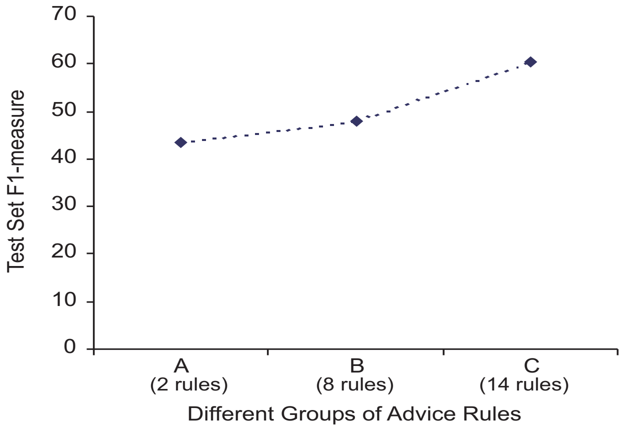

Figure 29: Disorder-Association Domain: F_1-measure versus Different Groups of Advice Rules on WAWA-IE with no Selector. Groups A, B, and C have 2, 8, and 14 rules, respectively. The curve represents WAWA-IE's performance without a selector during training and testing.

7.5 Summary

The main reason WAWA-IE performs so well is because (*i*) it has a recall bias, (*ii*) it is able to generate informative negative training examples (which are extremely important in the IE task since there are a lot of near misses in this task), and (*iii*) using prior domain knowledge, positive training examples, and its system-generated negative examples, it is able to improve on its precision after the learning process.

Chapter 8

Related Work

WAWA is closely related to several areas of research. The first is information retrieval and text categorization, the second is instructable software, and the third is information extraction. The following sections summarize the work previously done in each of these areas and relate it to the research presented in this dissertation.

8.1 Learning to Retrieve from the Web

WAWA-IE, Syskill and Webert (Pazzani, Muramatsu, and Billsus 1996), and WebWatcher (Joachims, Freitag, and Mitchell 1997) are Web agents that use machine learning techniques. Syskill and Webert uses a Bayesian classifier to learn about interesting Web pages and hyperlinks. WebWatcher employs a reinforcement learning and TFIDF hybrid to learn from the Web. Unlike WAWA-IR, these systems are unable to accept (and refine) advice, which usually is simple to provide and can lead to better learning than manually labeling many Web pages.

Drummond *et al.* (1995) have created a system which assists users browsing software libraries. Their system learns unobtrusively by observing users' actions. Letizia (Lieberman 1995) is a system similar to Drummond *et al.*'s that uses lookahead search from the current location in the user's Web browser. Compared to WAWA-IR, Drummond's system and Letizia are at a disadvantage since they cannot take advantage of advice given by the user.

WebFoot (Soderland 1997) is a system similar to WAWA-IR, which uses

HTML page-layout information to divide a Web page into segments of text. Wawa-IR uses these segments to extract input features for its neural networks and create an expressive advice language. WebFoot, on the other hand, utilizes these segments to extract information from Web pages. Also, unlike Wawa-IR, WebFoot only learns via supervised learning.

CORA (McCallum, Nigam, Rennie, and Seymore 1999; McCallum, Nigam, Rennie, and Seymore 2000) is a domain-specific search engine on computer science research papers. Like Wawa-IR, it uses reinforcement-learning techniques to efficiently spider the Web (Rennie and McCallum 1999; McCallum, Nigam, Rennie, and Seymore 2000). CORA's reinforcement learner is trained off-line on a set of documents and hyperlinks which enables its Q-function to be learned via dynamic programming, since both the reward function and the state transition function are known. Wawa-IR's training, on the other hand, is done on-line. Wawa-IR uses temporal-difference methods to evaluate the reward of following a hyperlink. In addition, Wawa-IR's reinforcement-learner automatically generates its own training examples and is able to accept and refine user's advice. CORA's reinforcement-learner is unable to perform either of these two actions. To classify text, CORA uses naive Bayes in combination with the EM algorithm (Dempster, Laird, and Rubin 1977), and a statistical technique named "shrinkage" (McCallum and Nigam 1998; McCallum, Rosenfeld, Mitchell, and Ng 1998). Again, unlike Wawa-IR, CORA's text classifier learns only through training examples and cannot accept and refine advice.

8.2 Instructable Software

Wawa is closely related to RATLE (Maclin 1995). In RATLE, a teacher continuously gives advice to an agent using a simple programming language. The advice specifies actions an agent should take under certain conditions. The agent learns by using connectionist reinforcement-learning techniques. In empirical results, RATLE outperformed agents which either do not accept advice or do not refine the advice.

Gordon and Subramanian (1994) use genetic search and high-level advice to refine prior knowledge. An advice rule in their language specifies a goal which will be achieved if certain conditions are satisfied.

Diederich (1989) and Abu-Mostafa (1995) generate examples from prior knowledge and mix these examples with the training examples. In this way, they indirectly provide advice to the neural network. This method of giving advice is restricted to prior knowledge (*i.e.*, it is not continually provided).

Botta and Piola (1999) describe a connectionist algorithm for refining numerical constants expressed in first-order logic rules. The initial values for the numerical constants are determined such that the prediction error of the knowledge base on the training set is minimized. Predicates containing numerical constants are translated into continuous functions, which are tuned by using the error gradient descent (Rumelhart and McClelland 1986). The advantage of their system, called the Numerical Term Refiner (NTR), is that the classical logic semantics of the rules is preserved. The weakness of their system is that, other than manipulating the numerical constants such that a predicate is always false or a literal from a clause is always true, they are not able to change the structure of the original knowledge base by incrementally adding new hidden units.

Fab (Balabanovic and Shoham 1997) is a recommendation system for the Web which combines techniques from content-based systems and collaborative systems. Page evaluations received from users are used to updated *Fab*'s search and selection heuristics. Unlike Wawa which has a collection of agents for just one user, *Fab* consists of a society of collaborative agents for a group of users. That is, in *Fab*, pages returned to one user are influenced by page ratings made by other users in the society. This influence can be viewed as an indirect form of advice because the agent for user A will adapt his behavior upon getting feedback on how the agent for a similar user B reacted to a page.

8.3 Learning to Extract Information from Text

I was unable to find any system in the literature that applies theory refinement to the IE task. Most IE systems break down into two groups. The first group uses some kind of relational learning to learn extraction patterns (Califf 1998; Freitag 1998b; Soderland 1999; Freitag and Kushmerick 2000). The second group learns parameters of hidden Markov models (HMMs) and uses the HMMs to extract information (Leek 1997; Bikel, Schwartz, and Weischedel 1999; Freitag and McCallum 1999; Seymore, McCallum, and Rosenfeld 1999; Ray and Craven 2001). In this section, I discuss some of the recently developed IE systems in both of these groups. I also review some systems that use IE to do IR and vice versa. Finally, I describe the *named-entity* problem, which is a task closely related to IE.

8.3.1 Relational Learners in IE

This section reports on four systems that use some form of relational learning to solve the IE problem. They are: (*i*) *RAPIER*, (*ii*) *SRV*, (*iii*) *WHISK*, and (*iv*) *BWI*.

RAPIER

RAPIER (Califf 1998; Califf and Mooney 1999), short for *Robust Automated Production of IE Rules*, takes as input pairs of training documents and their associated filled templates and learns extraction rules for the slots in the given template. RAPIER performs a specific-to-general (*i.e.*, bottom-up) search to find extraction rules.

Each extraction rule in RAPIER has three parts: (1) a pattern that matches the text immediately preceding the slot filler, (2) a pattern that matches the actual slot filler, and (3) a pattern that matches the text immediately following the slot filler. A pattern consists of a set of constraints on either one word or a list of words. Constraints are allowed on specific words, part-of-speech tags assigned to words, and the semantic-class of words.[1]

As mentioned above, RAPIER works bottom-up. For each training instance, it generates a rule which matches the target slots for that instance. Then, for each slot, pairs of rules are randomly selected and the least general generalization of the pair is found by performing a best-first beam search. The rules are sorted by using an information-gain metric, which prefers simpler rules. When the best scored rule matches all of the fillers in the training templates for a slot, the rule is added to the knowledge base. If the value of the best scored rule does not change across several successive iterations, then that rule is not picked for further modifications. The algorithm terminates if a rule is not added to the knowledge base after a specified threshold. RAPIER can only produce extraction patterns for single-slot extraction tasks.

SRV

Freitag (1998a, 1998b, 1998c) presents a multi-strategy approach to learn extraction patterns from text. The system combines a rote learner, a naive Bayes classifier, and a relational learner to induce extraction patterns.

The rote learner matches the phrase to be extracted against a list of correct slot fillers from the training set. The naive Bayes classifier estimates the probability that the terms in a phrase are in a correct slot filler. The hypothesis in this case has two parts: (1) the starting position of the slot filler and (2) the length of the slot filler. The priors for the position and length are determined by the training set. The naive Bayes classifier assumes that the position and the length of a slot are independent of each other. Therefore, it calculates the prior for a hypothesis to be the product of the two priors for a given position and length.

[1] Eric Brill's tagger (1994) is used to get part-of-speech tags and WordNet (Miller 1995) is used to get semantic-class information.

His relational learner (named SRV) is similar to FOIL (Quinlan 1990). It has a general-to-specific (top-down) covering algorithm. Each predicate in SRV belongs to one of the following five pre-specified predicates. The first is a predicate called `length`, which checks to see if the number of tokens in a fragment is less than, greater than, or equal to a given integer. The second is a predicate called `position`, which places a constraint on the position of a token in a rule. The third is a predicate called `relops`, which constrains the relative positions of two tokens in a rule. The fourth and fifth predicates are called `some` and `every`, respectively. They check whether a token's feature matches that of a user-defined feature (such as, capitalization, digits, and word length).

The system takes as input a set of annotated documents and does not require any syntactic analysis. However, it is able to use part-of-speech and semantic classes when they are provided. Freitag's system produces patterns for single-slot extraction tasks only.

WHISK

WHISK (Soderland 1999) uses active-learning techniques[2] to minimize the need for a human to supervise the system. That is, at each learning iteration, WHISK presents the user with a set of untagged examples which will convey the most information and lead to better coverage or accuracy of the evolving rule set. WHISK randomly picks the set of untagged examples from three categories: (1) examples that are covered by an existing rule, (2) examples that are near misses of a rule, and (3) examples that are not covered by any rule. The user defines the proportion of examples taken from each category. The default setting is $\frac{1}{3}$. The decision to stop giving training examples to WHISK is made by the user.

WHISK uses a general-to-specific (top-down) covering algorithm to learn

[2]Active-learning methods select untagged examples that are near decision boundaries. The selected examples are presented to and tagged by the user. The use of voting schemes or assignments of confidence levels to classifications are the most popular active learning methods.

extraction rules. Rules are created from a seed instance. A seed instance is a tagged-example which, is not covered by any of the rules in the rule set. Since WHISK does hill-climbing to extend rules, it is not guaranteed to produce an optimal rule.

WHISK represents extraction rules in a restricted form of regular expressions. It can produce rules for both single-slot and multi-slot extraction tasks. Moreover, WHISK is able to extract from structured, semi-structured, and free-text.

BWI

Freitag and Kushmerick (2000) combine wrapper induction techniques (Kushmerick 2000) with the AdaBoost algorithm (Schapire and Singer 1998) to create an extraction system named BWI (short for Boosted Wrapper Induction). Specifically, the BWI algorithm iteratively learns contextual patterns that recognize the *heads* and *tails* of slots. Then, at each iteration, it utilizes the AdaBoost algorithm to reweigh the training examples that were not covered by previous patterns.

Discussion

SRV and RAPIER build rules that individually specify an absolute order for the extraction tokens. WAWA-IE, WHISK, and BWI use wildcards in extraction rules to specify a relative order for the tokens.

WAWA-IE out-performs RAPIER, SRV and WHISK on the CMU seminar-announcement domain (see Section 7.1). BWI out-performed many of the relational learners and was competitive with systems using HMMs and WAWA-IE.

It is interesting to note that BWI has a bias towards high precision and tries to improve its recall measure by learning hundreds of rules. WAWA-IE, on the other hand, has a bias towards high recall and tries to improve its precision through learning about the extraction slots.

Only WAWA-IE and WHISK are able to handle multi-slot extraction tasks. The ability to accept and refine advice makes WAWA-IE less of a burden on the user than the other methods mentioned in this section.

8.3.2 Hidden Markov Models in IE

Leek (1997) uses HMMs for extracting information from biomedical text. His system uses a lot of initial knowledge to build the HMM model before using the training data to learn the parameters of HMM. However, his system is not able to refine the knowledge.

Freitag and McCallum (1999) use HMMs and a statistical method called "shrinkage"[3] to improve parameter estimation when only a small amount of training data is available. Each extraction field has its own HMM and the state-transition structure of the HMM is hand-coded. Recently, the same authors purposed an algorithm for automating the process of finding good structures for their HMMs (Freitag and McCallum 2000). Their algorithm starts with a simple HMM model and performs a hill-climbing search in the space of possible HMM structures. A move in the space is a split in a state of the HMM and the heuristic used is the performance of the resulting HMM on a validation set.

Seymore *et al.* (1999) use one HMM to extract many fields. The state-transition structure of the HMM is learned from training data. This extraction system is implemented in the CORA search engine (McCallum, Nigam, Rennie, and Seymore 2000) discussed in Section 8.1. They get around the problem of not having sufficient training examples by using data that is labeled for the information-retrieval task in their system.

Ray and Craven (2001) use HMMs to extract information from free text domains. They have developed an algorithm for incorporating grammatical

[3]Shrinkage tries to find a happy medium between the size of the training data and the number of states used in a HMM.

structure of sentences in an HMM and have found that such information improves performance. Moreover, instead of training their HMMs to maximize the likelihood of the data given the model, they maximize the likelihood of predicting correct sequences of slot fillers. This objective function has also improved their performance.

Discussion

Wawa-IE and the HMM produced by Freitag and McCallum (2000) are competitive in the CMU seminar-announcement data set. Wawa-IE was able to out-perform Ray and Craven's HMMs on the two biomedical domains described in Section 7.

Wawa-IE has three advantages over the systems that use HMM. The first advantage of Wawa-IE is that it is able to utilize and refine prior knowledge, which reduces the need for a large number of labeled training examples. However, Wawa-IE does not depend on the initial knowledge being 100% correct (due to its learning abilities). I believe that it is relatively easy for users to articulate some useful domain-specific advice (especially when a user-friendly interface is provided that converts their advice into the specifics of WAWA's advice language). The second advantage of Wawa-IE is that the entire content of the document is used to estimate the correctness of a candidate extraction. This allows Wawa-IE to learn about the extraction slots and the documents in which they appear. The third advantage of WAWA-IE is that it is able to utilize the untrained ScorePage network to produce some informative negative training examples (i.e., near misses), which are usually not provided in IE tasks.

8.3.3 Using Extraction Patterns to Categorize

Riloff and Lehnert (1994, 1996) describe methods for using the patterns generated by an information-extraction system to classify text. During training, a signature is produced by pairing each extraction pattern with the words in the training set that satisfy that pattern. This signature is then labeled as a relevancy signature if it is highly correlated with the relevant documents in the training set. In the testing phase, a document is labeled as relevant only if it contains one of the generated relevancy signatures.

There is a big difference between the advice rules given by the user to WAWA and the extraction patterns generated by the information-extraction systems used in Riloff and Lehnert (1994) and Riloff (1996). WAWA's advice rules define the behavior of an IE agent upon encountering documents. The extraction patterns generated in Riloff and Lehnert's work are learned from a set of labeled training examples and are used to extract words and phrases from sentences.

8.3.4 Using Text Categorizers to Extract

Only a few researches have investigated using text classifiers to solve the extraction problem. Craven and Kumlien (1999) use a sentence classifier to extract instances of the subcellular-localization relation from an earlier version of the subcellular-localization data set described in Section 7.2. Specifically, if $r(X, Y)$ is the target binary relation, then the goal is to find instances x and y where x and y are in the semantic lexicons of X and Y respectively. For example, the binary relation *cell-localization(protein, cell-type)* describes the cell types in which a particular protein is located.

Craven and Kumlien (1999) use the naive Bayes algorithm with a bag-of-words representation (Mitchell 1997) to classify sentences. A sentence is classified as a positive example if it contains at least one instance of the target relation. Then, to represent linguistic structure of documents, they learned IE rules by using a relational learning algorithm similar to FOIL (Quinlan 1990). Craven and Kumlien (1999) use "weakly" labeled training data to reduce the need for labeled training examples.

WAWA-IE is similar to Craven and Kumlien's system in that I use essentially a text classifier to extract information. However, in Craven and Kumlien (1999), the text classifier processes small chunks of text (such as sentences) and extracts binary relations of the words that are in the given semantic lexicons. In WAWA-IE, the text classifier is able to classify a text document as a whole and generates a lot of extraction candidates without the need for semantic lexicons. Another difference between WAWA-IE and Craven and Kumlien's system is that their text classifier is built for the sole purpose of extracting information. However, WAWA was initially built to classify text documents (of any size) and its extraction ability is a side effect of the way the classifier was implemented.

Zaragoza and Gallinari (1998) use hierarchical information-retrieval methods to reduce the amount of data given to their stochastic information-extraction system. The standard IR technique of TF/IDF weighting (see Section 2.4) is used to eliminate irrelevant documents. Then, the same IR process is used to eliminate irrelevant paragraphs from relevant documents. Relevant paragraphs have at least one extraction template associated with them. Finally, Hidden Markov Models are used to extract patterns at the word level from a set of relevant paragraphs. WAWA-IE is different than Zaragoza and Gallinari's system in that I do not use a text classifier to filter out data and then use another model to extract data. A WAWA-IE agent is trained to directly extract by rating an extraction within the context of the entire document.

8.3.5 The Named-Entity Problem

A task close to IE is the *named-entity* problem. The named-entity task is the problem of recognizing all names (*i.e.*, people, locations, and organizations), dates, times, monetary amounts, and percentages in text. I have come across two learning systems that focus on extracting names.

The first is $IdentiFinder^{TM}$ (Bikel, Schwartz, and Weischedel 1999), which uses a Hidden Markov Model and textual information (such as capitalization and punctuation) to learn to recognize and classify names, dates, times, and numerical quantities. A name is classified into three categories: the name of a person, the name of a location, and the name of an organization. Numerical quantities are classified into monetary amounts or percentages. $IdentiFinder^{TM}$ is independent of the case of the text (*i.e.*, all lower-case, all capitalized, or mixed) and was applied to text in English and Spanish.

In the second system, Baluja *et al.* (1999) use a decision-tree classifier in conjunction with information from part-of-speech tagging, dictionary lookup, and textual information (such as capitalization) to extract names. Their system does not attempt to distinguish between names of persons, locations, and organizations.

WAWA-IE does not use dictionary lookups as evidence. One advantage of my system compared to $IdentiFinder^{TM}$ and Baluja *et al.*'s system is that I do not need a large number of labeled training examples to achieve high performance. This is because I address the problem by using theory-refinement techniques, which allow users to easily provide task-specific information via approximately correct inference rules (e.g., WAWA's advice rules).

8.3.6 Adaptive and Intelligent Sampling

The interest in being able to *intelligently* search a large space is not new. However, one of the recent advances in this area was done by Boyan and Moore (1998, 2000). They describe an intelligent search algorithm called STAGE, which utilizes features of the search space[4] to predict the outcome of a local search algorithm (*e.g.*, hillclimbing or WalkSAT). In particular, STAGE learns an evaluation function for a specific problem and uses it to bias future search paths toward optima in the search space. More recently, Boyan and Moore (2000) present an algorithm, called XSTAGE, which is able to utilize previously

[4]An example of a feature used in STAGE is the number of bins in the bin-packing problem (Coffman, Garey, and Johnson 1996).

learned evaluation functions on new and similar search problems. WAWA-IE's selector use the trained SCOREPAGE network to find good search trajectories.

Chapter 9

Conclusions

In this thesis, I argue that a promising way to create effective and intelligent IR and IE agents is to involve both the user's ability to do direct programming (*i.e.*, provide approximately correct instructions of some sort) along with the agent's ability to accept and automatically create training examples. Due to the largely unstructured nature and the size of on-line textual information, such a hybrid approach is more appealing than ones solely based on either non-adaptive agent programming languages or large amounts of annotated examples.

This chapter presents the contributions of my thesis along with its limitations and some ideas about future work.

9.1 Contributions

The underlying contribution of this thesis is in its use of theory-refinement techniques for the information retrieval and information extraction tasks. The specific contributions of this thesis to each task are listed below.

Wawa-IR

The significant contribution of Wawa's IR system is in its ability to utilize the following three sources of information when learning to convert from a general search engine to a personalized and specialized IR agent:

1. users' prior and continual instruction
2. users' manually labeled (Web) pages

3. system-measured feedback from the Web

The advantages of utilizing users' prior and continual instruction are seen in two ways: (*i*) the agent performs reasonably well initially and (*ii*) the agent does not need to process a large number of training examples to specialize in the desired IR task. However, if labeled examples become available, then Wawa-IR can use them during the agent's training process.

Wawa-IR utilizes the feedback it gets from the Web by using temporal-difference methods (that are based on beam-search as opposed to hill climbing) to evaluate the reward of following a hyperlink. In this way, Wawa-IR's reinforcement-learner automatically generates its own training examples and is able to accept and refine users' advice.

Wawa-IE

The five major contributions of Wawa's IE system are as follows:

1. By using theory refinement, it is able to reduce the need for large number of annotated examples (which are very expensive to obtain in extraction tasks).

2. By using the untrained neural network (which only contains user's initial advice), it is able to create informative negative training examples (which are rare in IE tasks) .

3. By using an IR agent to score each possible extraction, it uses the content of the entire document to learn about extraction template

4. By using a generate-and-test approach to information extraction, Wawa-IE has a bias towards high recall. It improves its precision on the IE task during the training process. The bias towards high recall plus the improvement on precision lead to high F_1-measures on Wawa-IE's trained agents.

5. By using heuristically inclined sampling techniques, Wawa-IE has been able to reduce the computational burden inherent in generate-and-test approaches.

9.2 Limitations and Future Directions

The main limitations of Wawa are: (*i*) speed and (*ii*) comprehensibility. One cost of using Wawa-IR is that I fetch and analyze many Web pages. In this thesis, I have not focused on speed of an IR agent, ignoring such questions as how well does an Wawa-IR agent perform if it only used the capsule summaries that the search engines return, etc. Making Wawa-IR faster is part of future directions.

Due to my use of artificial neural networks, it is difficult to understand what was learned (Craven and Shavlik 1996). It would be nice if a Wawa-IR agent could explain its reasoning to the user. In an attempt to alleviate this problem, I have built a "visualizer" for each neural network in Wawa-IR. The visualizer draws the neural network and graphically displays information on all nodes and links in the network.

The main directions of future work are as follows:

1. Better understand what people would like to say to an instructable Web agent such as Wawa and improve Wawa's advice language accordingly.

2. Embed Wawa into a major, existing Web browser, thereby minimizing new interface features that users must learn in order to interact with my system.

3. Develop methods whereby Wawa can automatically infer plausible training examples by observing users' normal use of their browsers (Goecks and Shavlik 2000).

4. Incorporate the candidate generation and selection steps directly into WAWA's connectionist framework, whereby the current SCOREPAGE network would find new candidate extractions during the training process.

5. Learn to *first* reduce the number of candidates per slot and then apply WAWA-IE's selectors to the reduced list of individual-slot extraction candidates.

6. Test WAWA-IE's scaling abilities (*e.g.* run WAWA-IE on a combination-slots extraction task that has more than two extraction slots).

7. Use other supervised learning algorithms such as support vector machines (Cristianini and Shawe-Taylor 2000) or hidden Markov models (Rabiner and Juang 1993) as WAWA's supervised learner. Moreover, explore ways in which theory-refinement techniques can be applied to these other algorithms.

9.3 Final Remarks

At this moment in time, the task of retrieving and extracting information from on-line documents is like trying to find a couple of needles in a colossal haystack. When encountered with such hard problems, one should utilize all of the available resources. Such is the case in the system that I have presented in this dissertation. I hope that the experimental successes of WAWA will persuade more researchers to investigate theory-refinement approaches in their learning systems.

Appendix A

WAWA's Advice Language

This appendix presents WAWA's advice language and is divided into four segments. The first segment introduces the sentences in WAWA's advice language, which are conditional statements (*i.e.*, *if condition then action*). The second and third segments list the conditions and the actions in WAWA's advice language, respectively. The last segment describes the different user-interfaces for giving advice.

Sentences

A sentence in WAWA's advice language has one of the following forms:

S1. WHEN `condition` THEN `action`

This statement advises the agent that when the condition is true, it should pursue the given action.

S2. *ScaleLinearlyBy*(`conditions`) THEN `action`

In this statement, the agent is advised to consider the conditions listed in the *ScaleLinearlyBy* function. The *ScaleLinearlyBy* function takes a set of conditions and returns a number between 0 and 1, proportional to the number of conditions satisfied. The conditions are given the same weights inititally; however these weights can change during training. The activation function of the hidden unit created by this type of advice is linear.

S3. WHEN `guarded_condition` *ScaleLinearlyBy*(`conditions`) THEN `action`

This statement advises the agent to consider the conditions listed in the *ScaleLinearlyBy* function and consequently pursue the given action only if the `guarded_ condition` is true.

Conditions

The conditions used in advice rules are presented in BNF, short for *Backus-Naur Form*, notation (Aho, Sethi, and Ullman 1986). The non-terminals are bold face. The terminals are surrounded by quotation marks. Predicates and functions are italicized. Parameters are in typewriter font.

conditions →
  condition |
  conditions |
  condition **connective** **conditions** |
  (**conditions**) **connective** **conditions**
connective →
  "AND" | "And" | "and" | "&" | "&&" | "OR" | "Or" | "or" | "|" | "||"

Among the predicates listed under the non-terminal **condition**, *NofM* looks for (Web) pages where at least N of the M given conditions are true.

condition →
  ["NOT" | "not" | "Not"] **condition** |
  NofM(`integer`, **conditions**) |
  word_locations_constructs |
  phrase_constructs |
  nearness_constructs |
  general_features |
  numeric_operations

The predicate *anyOf* is true if *any* of the listed words is present. The predicate *noneOf* is true if *none* of the listed words is present. The *anyOf* and *noneOf* predicates may be used in conditions where one or more words are required.

terms →
  `word` | `words` | *anyOf*(`words`) | *noneOf*(`words`) |
  terms "·" **terms** | **terms** "∗" **terms**

The predicates listed under **word_locations_constructs** look for words in specific locations on a Web page. Here is an explanation of some of the non-obvious predicates for this group. The predicate *anywhereInSection* is true if the listed terms are found in the given section depth relative to the current nesting of the sliding window. The value of *relative_section_depth* must be either 1 (for the parent section), 2 (for the grandparent section), or 3 (for the great-grandparent section).

The predicates *anywhereInLeftSideOfWindow* and *anywhereInRightSideOfWindow* are true if the given terms are in the bags to the left and right of the current window position, respectively.

The predicates *atBackPosInHyperlinkHost* and *atBackPosInUrlHost* determine whether the given word is found at the specified position from the end of the hyperlink's or URL's host, respectively. For example, `atBackPosInHyperlinkHost(1, edu)` looks for hyperlinks that have "edu" as the last word of their host (*e.g.*, `www.wisc.edu`).

The predicates *atLeftSpotInWindow* and *atRightSpotInWindow* are true when the given word is at the specified position to the left or right of the center of the sliding window, respectively.

word_locations_constructs →
anywhereOnPage(**terms**) |
anywhereInTitle(**terms**) |
anywhereInUrlHost(**terms**) |
anywhereInUrlNonHost(**terms**) |
anywhereInSections(**terms**) |
anywhereInCurrentSection(**terms**) |
anywhereInSection(`relative_section_depth`, **terms**) |
anywhereInHyperlinkHost(**terms**) |
anywhereInHyperlinkNonHost(**terms**) |
anywhereInHypertext(**terms**) |
anywhereInLeftSideOfWindow(**terms**) |

anywhereInWindow(**terms**) |
anywhereInRightSideOfWindow(**terms**) |
titleStartsWith(**terms**) |
titleEndsWith(**terms**) |
urlHostStartsWith(**terms**) |
urlHostEndsWith(**terms**) |
urlNonHostStartsWith(**terms**) |
urlNonHostEndsWith(**terms**) |
currentSectionStartsWith(**terms**) |
currentSectionEndsWith(**terms**) |
hyperlinkHostStartsWith(**terms**) |
hyperlinkHostEndsWith(**terms**) |
hyperlinkNonHostStartsWith(**terms**) |
hyperlinkNonHostEndsWith(**terms**) |
atLeftSpotInTitle(`position`, `word`) |
atRightSpotInTitle(`position`, `word`) |
atBackSpotInURLHost(`position`, `word`) |
atLeftSpotInURL(`position`, `word`) |
atRightSpotInURL(`position`, `word`) |
atLeftSpotInCurrentSection(`position`, `word`) |
atRightSpotInCurrentSection(`position`, `word`) |
atBackSpotInHyperlinkHost(`position`, `word`) |
atLeftSpotInHyperlink(`position`, `word`) |
atRightSpotInHyperlink(`position`, `word`) |
atLeftSpotInWindow(`position`, `word`) |
atCenterOfWindow(`word`) |
atRightSpotInWindow(`position`, `word`) |
POSatLeftSpotInWindow(`position`, **part_of_speech_tag**) |
POSatCenterOfWindow(**part_of_speech_tag**) |
POSatRightSpotInWindow(`position`, **part_of_speech_tag**)

The predicates listed under **phrase_constructs** determine where a phrase is present on a page or in localized parts of it (*e.g.*, the page's title). The predicate *consecutiveInTitle* is true if the given phrase is in the title of the page. The predicates *consecutiveInUrlHost* and *consecutiveInUrlNonHost* determine if the specified phrase is in the host or the non-host segments of the page's URL, respectively. The predicates *consecutiveInHyperlinkHost* and *consecutiveInHyperlinkNonHost* are similar to *consecutiveInUrlHost* and *consecutiveInUrlNonHost* except that they look for the given phrase in the host or the non-host segments of the page's hyperlinks, respectively.

phrase_constructs →
consecutive(**terms**) |
consecutiveInTitle(**terms**) |
consecutiveInUrlHost(**terms**) |
consecutiveInUrlNonHost(**terms**) |
consecutiveInaSection(**terms**) |
consecutiveInHyperlinkHost(**terms**) |
consecutiveInHyperlinkNonHost(**terms**) |
consecutiveInHypertext(**terms**)

The predicates listed under **nearness_constructs** can be divided into the following four subgroups:

1. *nearbyIn*`LOCATION` predicates, which are true when the given terms are near each other (*e.g.*, nearybyInTitle)

2. *nearbyIn*`LOCATION`*InOrder* predicates, which are true when the given terms are in order and near each other (*e.g.*, nearbyInTitleInOrder)

3. *nearbyIn*`LOCATION`*Within* predicates, which are true when the given terms are within the specified distance of each other (*e.g.*, nearbyInTitleWithin)

4. *nearbyIn*`LOCATION`*InOrderWithin* predicates, which are true when the given terms are in order and within the specified distance of each other (*e.g.*, nearbyInTitleInOrderWithin)

nearness_constructs →

nearby(**terms**) |
nearbyInTitle(**terms**) |
nearbyInUrlHost(**terms**) |
nearbyInUrlNonHost(**terms**) |
nearbyInSection(**terms**) |
nearbyInHostHyperlink(**terms**) |
nearbyInNonHostHyperlink(**terms**) |
nearbyInHypertext(**terms**) |
nearbyInOrder(**terms**) |
nearbyInTitleInOrder(**terms**) |
nearbyInUrlHostInOrder(**terms**) |
nearbyInUrlNonHostInOrder(**terms**) |
nearbyInSectionInOrder(**terms**) |
nearbyInHyperlinkHostInOrder(**terms**) |
nearbyInHyperlinkNonHostInOrder(**terms**) |
nearbyInHypertextInOrder(**terms**) |
nearbyWithin(`distance`, **terms**) |
nearbyInTitleWithin(`distance`, **terms**) |
nearbyInUrlHostWithin(`distance`, **terms**) |
nearbyInUrlNonHostWithin(`distance`, **terms**) |
nearbyInSectionWithin(`distance`, **terms**) |
nearbyInHyperlinkHostWithin(`distance`, **terms**) |
nearbyInHyperlinkNonHostWithin(`distance`, **terms**) |
nearbyInHypertextWithin(`distance`, **terms**) |
nearbyInOrderWithin(`distance`, **terms**) |
nearbyInTitleInOrderWithin(`distance`, **terms**) |
nearbyInUrlHostInOrderWithin(`distance`, **terms**) |
nearbyInUrlNonHostInOrderWithin(`distance`, **terms**) |
nearbyInSectionInOrderWithin(`distance`, **terms**) |

nearbyInHyperlinkHostInOrderWithin(`distance`, **terms**) |
nearbyInHyperlinkNonHostInOrderWithin(`distance`, **terms**) |
nearbyInHypertextInOrderWithin(`distance`, **terms**)

Among the predicates listed under **general_features**, *isaQueryWord* is true if the given word was in the user's search query. The predicates *queryWordAtLeftSpotInWindow*, *queryWordAtCenterOfWindow*, and *queryWordAtRightSpotInWindow* determine whether one of the user's query words is at a specific location in the sliding window.

general_features →
isaQueryWord(`word`) |
queryWordAtLeftSpotInWindow(`position`) |
queryWordAtCenterOfWindow |
queryWordAtRightSpotInWindow(`position`) |
insideTitle |
insideURL |
insideSection |
insideHypertext |
insideHyperlink |
insideForm |
insideFrameSet |
insideTable |
insideImageCaption |
insideEmailAddress |
insideMetaWords |
insideEmphasized |
insideAddress |
knownWhenPageExpires |
knownWhenLastModified

Among the predicates listed under **numeric_valued_features**, *fractPositionOnPage* determines the ratio of the position of the word in the center of window to the total number of words on the page.

numeric_operations →
  numeric_valued_features **numeric_operators** **numeric_valued_features** |
  numeric_valued_features **numeric_operators** `integer`
numeric_operators → "$<$" | "$\leq$" | "$==$" | "$>$" | "$\geq$"
numeric_valued_features →
  numberOfImagesOnPage |
  numberOfURLsOnPage |
  numberOfTablesOnPage |
  numberOfFormsOnPage |
  numberOfFramesOnPage |
  numberOfQueryWordsOnPage |
  numberOfQueryWordsInTitle |
  numberOfQueryWordsInURL |
  numberOfQueryWordsInCurrentSectionTitle |
  numberOfQueryWordsInCurrentHyperlink |
  numberOfQueryWordsInWindow |
  numberOfQueryWordsInLeftSideOfWindow |
  numberOfQueryWordsInRightSideOfWindow |
  numberOfWordsOnPage |
  numberOfWordsInTitle |
  numberOfFieldsInUrl |
  numberOfWordsInCurrentSectionTitle |
  numberOfFieldsInCurrentHyperlink |
  numberOfWordsInWindow |
  numberOfWordsInLeftSideOfWindow |
  numberOfWordsInRightSideOfWindow |
  numberOfWordsInSectionTitle(`integer`) |
  numberOfQueryWordsInSectionTitle(`integer`) |

fractPositionOnPage |
pageExpiresAt |
pageLastModifiedAt

part_of_speech_tag →
"commonNoun" | "properNoun" | "pluralNoun" | "pluralProperNoun" |
"baseFormVerb" | "pastTenseVerb" |
"presentParticipleVerb" | "pastParticipleVerb" |
"nonThirdPersonSingPresentVerb" | "thirdPersonSingPresentVerb" |
"otherAdjective" | "comparativeAdjective" | "superlativeAdjective" |
"otherAdverb" | "comparativeAdverb" | "superlativeAdverb" |
"hyphenatedWord" | "hyphenDroppedAfter" | "hyphenDroppedBefore" |
"allCaps" | "leadingCaps" | "title" | "cardinalNumber" | "punctuation" |
"personalPronoun" | "possessivePronoun" | "preposition" |
"determiner" | "coordinatingConjuntion" | "existentialThere" |
"whDeterminer" | "whPronoun" | "possessiveWhPronoun" | "whAdverb" |
"foreignWord" | "unknownWord"

Actions

The actions used in advice rules are as follows:

A1. *suggest doing both*
This action adds a moderately weighted link from the rule's new hidden unit in both the ScorePage and the ScoreLink networks into the output units of these networks.

A2. *suggest showing page*
This action adds a moderately weighted link from the rule's new hidden unit in the ScorePage network into the network's output unit.

A3. *suggest following link*
This action adds a moderately weighted link from the rule's new hidden unit in the ScoreLink network into the network's output unit.

A4. *avoid both*

This action adds a link with a moderately negative weight from the rule's new hidden unit in both the ScorePage and the ScoreLink networks into the output units of these networks.

A5. *avoid showing page*

This action adds a link with a moderately negative weight from the rule's new hidden unit in the ScorePage network into the network's output unit.

A6. *avoid following link*

This action adds a link with a moderately negative weight from the rule's new hidden unit in the ScoreLink network into the network's output unit.

Actions can be prefixed by the following four modifiers:

- *Definitely*: Assuming the conditions of a 'definite' rule are fully met, the link out of the sigmoidal hidden unit representing the rule will have a weight of
  - ◇ 11.25, for actions A1 through A3, and
  - ◇ -11.25, for actions A4 through A6.
- *Strongly*: Assuming the conditions of a 'strong' rule are fully met, the link out of the sigmoidal hidden unit representing the rule will have a weight of
  - ◇ 7.5, for actions A1 through A3, and
  - ◇ -7.5, for actions A4 through A6.
- *Moderately*: Assuming the conditions of a 'moderate' rule are fully met, the link out of the sigmoidal hidden unit representing the rule will have a weight of
  - ◇ 2.5, for actions A1 through A3, and
  - ◇ -2.5, for actions A4 through A6.

  When an action does not have modifier, then "moderately" is used as the default modifier.

- *Weakly*: Assuming the conditions of a 'weak' rule are fully met, the link out of the sigmoidal hidden unit representing the rule will have a weight of
  - ◇ 0.75, for actions A1 through A3, and
  - ◇ -0.75, for actions A4 through A6.
- *With zero-valued weights*: Assuming the conditions of a 'zero-weight' rule are fully met, the link out of the sigmoidal hidden unit representing the rule will have a weight of 0 for actions A1 through A6.

Advice Interfaces

The user can give advice to WAWA through three different interfaces: (*i*) the "basic" interface, (*ii*) the "intermediate" interface, and (*iii*) the "advanced" interface. All three of these interfaces are menu-driven and differ only in the number of advice constructs that they offer. The user can also compose her advice in her favorite text editor and give a text file to WAWA.

WAWA's advice parser is quite user-friendly by forgiving many of the user's syntactical mistakes. For example, the advice parser does not require the word "THEN" before an action and accepts the word "IF" as opposed to "WHEN" in advice rules. Also, when a major syntactical error is made, WAWA's advice parser just ignores the "bad" rule and proceeds with normal operations.

Finally, since advice files can get long, WAWA's advice language allows the user to annotate advice by placing comments inside brackets or after the # symbol.

Appendix B

Advice Used in the Home-Page Finder

This appendix presents the rules used to create the home-page finder from the generic WAWA-IR (see Chapter 5 for details on this case study). These rules contain the following 13 variables:

1. *?FirstName* ← First name of a person (*e.g.*, Robert)
2. *?FirstInitial* ← First initial of a person (*e.g.*, R)
3. *?NickNameA* ← First nickname of a person (*e.g.*, Rob)
4. *?NickNameB* ← Second nickname of a person (*e.g.*, Bob)
5. *?MiddleName* ← Middle name of a person (*e.g.*, Eric)
6. *?MiddleInitial* ← Middle initial of a person (*e.g.*, E)
7. *?LastName* ← Last name of a person (*e.g.*, Smith)
8. *?MiscWord1* ← First miscellaneous word
9. *?MiscWord2* ← Second miscellaneous word
10. *?MiscWord3* ← Third miscellaneous word
11. *?UrlHostWord1* ← Third word from the end of a host url (*e.g.*, `cs` in `http://www.cs.wisc.edu`)
12. *?UrlHostWord2* ← Second word from the end of a host url (*e.g.*, `wisc` in `http://www.cs.wisc.edu`)
13. *?UrlHostWord3* ← Last word in a host (*e.g.*, `edu` in `http://www.cs.wisc.edu`)

In my experiments, I only used variables numbered 1 through 7 since I wanted to fairly compare WAWA's home-page finder to existing alternative approaches. That is, I did not provide any values for variables numbered 8 through 13.

I introduced these variables and even wrote some rules about them only to illustrate that there is other information besides a person's name that might be helpful in finding his/her home-page.

The syntax and semantics of the advice language used below are described in Appendix A. Before I present my home-page finder rules, I will define some non-terminal tokens (used in my rules) in Backus-Naur form (Aho, Sethi, and Ullman 1986). These non-terminal tokens are:

first_names	→	*?FirstName* | *?FirstInitial* | *?NicknameA* | *?NicknameB*
middle_names	→	*?MiddleName* | *?MiddleInitial*
full_name	→	**first_names middle_names** *?LastName*
regular_name	→	**first_names** *?LastName*
home_page_words	→	home | homepage | home-page
person	→	**full_name** | **regular_name**

The following rules look for *pages* with the person's name and either the phrase "home page of" or the words "home", "homepage", or "home-page" in the title. Recall that the function *consecutiveInTitle* takes a sequences of words and returns true if they form a phrase inside the title of a page. The symbol "**.**" is WAWA's "wild card" symbol. It is a placeholder that matches any single word or punctuation.

home_page_rules_A →

WHEN consecutiveInTitle("home page of" **person**)
THEN definitely suggest showing page |

WHEN consecutiveInTitle(**person** "'s" **home_page_words**)
THEN definitely suggest showing page |

WHEN consecutiveInTitle(**person home_page_words**)
THEN strongly suggest showing page |

WHEN consecutiveInTitle(
first_names . *?LastName* "'s" **home_page_words**)
THEN suggest showing page |

WHEN consecutiveInTitle(
"home page of" **first_names .** *?LastName*)
THEN suggest showing page |

WHEN consecutiveInTitle("home page of" **.** *?LastName*)
THEN suggest showing page |

WHEN consecutiveInTitle("home page of" **. .** *?LastName*)
THEN weakly suggest showing page |

WHEN consecutiveInTitle(**person . home_page_words**)
THEN suggest showing page

The next set of rules look for *links* to the person's home-page. Recall that the function *consecutive* takes a sequences of words and returns true if the words appear as a phrase on the page.

home_page_rules_B $\rightarrow$

WHEN consecutive(**person** "'s" **home_page_words**)
THEN definitely suggest following link |

WHEN consecutive("home page of" **person**)
THEN definitely suggest following link |

WHEN consecutive(**first_names .** *?LastName* "'s" **home_page_words**)
THEN strongly suggest following link |

WHEN consecutive("home page of" **first_names .** *?LastName*)
THEN strongly suggest following link |

WHEN consecutive("home page of" **.** *?LastName*)
THEN suggest following link |

WHEN consecutive(**person**)
THEN strongly suggest following link |

WHEN consecutive(*?LastName*)
THEN suggest following link

The following rules look for *pages* and *links leading to pages* with the person's name in the title, but not in a question format. I do not want both the person's name and a question mark in a page's title because it could represent a query on that person and not the person's home-page.

home_page_rules_C →

WHEN (NOT(anywhereInTitle("?")) AND
    consecutiveInTitle(**regular_name**
        noneOf(**home_page_words**)))
THEN strongly suggest doing both |

WHEN (NOT(anywhereInTitle("?")) AND
    consecutiveInTitle(**first_names** . *?LastName*
        noneOf(**home_page_words**)))
THEN strongly suggest doing both |

WHEN (NOT(anywhereInTitle("?")) AND
    consecutiveInTitle(**first_names**) AND
    anywhereInTitle(*?LastName*))
THEN suggest doing both |

WHEN (NOT(anywhereInTitle("?")) AND
    consecutiveInTitle(*?LastName* "," **first_names**))
THEN suggest doing both |

WHEN consecutive(**first_names** "'s" **home_page_words**)
THEN suggest doing both |

WHEN consecutive(*?LastName* **home_page_words**)
THEN suggest doing both |

WHEN consecutive("my" **home_page_words**)
THEN suggest doing both

This next rule looks for *home-pages* that might lead to other home-pages.

home_page_rules_D $\rightarrow$

WHEN (NOT(anywhereInTitle("?")) AND
(anywhereInTitle("home page") OR
anywhereInTitle("home-page") OR
anywhereInTitle("homepage")))
THEN suggest following link

The rule below seeks *pages* that have the person's last name near an image. I conjecture that the image might be that person's picture.

home_page_rules_E $\rightarrow$

WHEN (insideImageCaption() AND consecutive(*?LastName*))
THEN suggest doing both

The next set of rules look for *pages* and *links* that include some of the query words given by the user (*i.e.*, bindings for some of the variables). These rules use functions like *numberOfQueryWordsOnPage*, which (obviously) returns the number of query words on the page. This set also includes rules that check for URLs and hyperlinks containing the "?" symbol. Such URLs and hyperlinks point to pages generated by search engines (*aka*, query pages) and should be avoided.

home_page_rules_F $\rightarrow$

WHEN ((insideEmailAddress() OR insideAddress()) AND
(numberOfQueryWordsInWindow() $\geq$ 1))
THEN weakly suggest doing both |

WHEN (numberOfQueryWordsOnPage() $<$ 1)
THEN avoid following link AND definitely avoid showing page |

WHEN anywhereInURL("?")
THEN strongly avoid following link & definitely avoid showing page |

WHEN anywhereInCurrentHyperlink("?")
THEN strongly avoid following link |

WHEN (anywhereInURL($\sim$) AND NOT(anywhereInURL("?"))
AND (numberOfQueryWordsInURL() ≥ 1))
THEN weakly suggest both

The next set of rules look for *pages* and *links* that include some of the query words given by the user. They use a function called *ScaleLinearlyBy* which takes a set of conditions and returns a number between 0 and 1, proportional to the number of conditions satisfied (see Appendix A for details).

home_page_rules_G $\rightarrow$

ScaleLinearlyBy(numberOfQueryWordsInWindow())
THEN suggest both |

ScaleLinearlyBy(numberOfQueryWordsInTitle())
THEN suggest showing page AND weakly suggest following link

The following rules look for *pages* and *links* that include a combination of the words "home page," "home-page," "homepage," "home," "page," "directory," and "people" either on the page itself or in its URL.

home_page_rules_H $\rightarrow$

WHEN consecutive(**home_page_words** "directory")
THEN strongly suggest following link |

WHEN consecutiveInaSection(**home_page_words** "directory")
THEN suggest following link |

WHEN consecutiveInHyperText(**home_page_words**)
THEN suggest following link |

WHEN consecutiveInaSection(**home_page_words**)
THEN weakly suggest doing both |

WHEN (consecutive("home page") OR
consecutive(anyOf("homepage" "home-page")))
THEN weakly suggest doing both |

WHEN (anywhereInURL("home") OR anywhereInURL("page") OR
anywhereInURL("people") OR anywhereInURL("homepage") OR
anywhereInURL("home-page"))
THEN suggest doing both

The subsequent two rules attempt to find *when a page was last modified.* The function *pageLastModifiedAt*() determines the number of days from today since the page was modified. If the page does not specify when it was last modified, the value for the function *pageLastModifiedAt*() is zero. If the page reports the last time it was modified, I convert the time into the interval [0, 1], where 0 means never and 1 means the page was changed today. Then, the value of *pageLastModifiedAt*() is 1 minus the scaled value.

home_page_rules_I →

ScaleLinearlyBy(pageLastModifiedAt())
THEN weakly suggest showing page AND very weakly suggest following link

The next three rules use the function *consecutiveInURL*. As described in Appendix A, this function takes a sequences of words and returns true if the words form a phrase in the URL of the page. These rules look for *pages* that have a URL containing the person's name and possibly the words "htm" or "html." In a URL, when a login name is prefixed with ∼, it usually stands for the given user's home directory.

home_page_rules_J →

WHEN consecutiveInURL("∼" anyOf(**first_names** *?LastName*))
THEN weakly suggest doing both |

WHEN consecutiveInURL(**person** anyOf("htm" "html"))
THEN definitely suggest showing page

The following five rules look for *pages* and *links* that have the phrase "*?First-Name ?LastName*" anywhere on them or in their title, their URL, one of their hypertexts, or one of their hyperlinks.

home_page_rules_K $\rightarrow$

WHEN consecutive(*?FirstName ?LastName*)
THEN strongly do both |

WHEN consecutiveInURL(*?FirstName ?LastName*)
THEN strongly do both |

WHEN consecutiveInTitle(*?FirstName ?LastName*)
THEN strongly do both |

WHEN consecutiveInHypertext(*?FirstName ?LastName*)
THEN strongly do both |

WHEN consecutiveInHyperlink(*?FirstName ?LastName*)
THEN strongly do both

The next three rules avoid *pages* that have the phrase "404 not found" in their title.

home_page_rules_L $\rightarrow$

WHEN titleStartsWith(anyOf("404" "file") "not found")
THEN strongly avoid both |

WHEN titleEndsWith(anyOf("404" "file") "not found")
THEN strongly avoid both |

WHEN anywhereInTitle("404 not found")
THEN avoid both

The following rules contain advice about commonly used words on a person's homepage like "cv", "resume", etc. The function *NofM* used in this set of rules takes an integer, N and a list of conditions of size M. It returns true if at least N of the M conditions are true. The function $anywhereOnPage(\text{“}555-1234\text{”})$

is true if a telephone number is on the page. Otherwise, it returns false. These rules were removed from the original set of 76 advice rules to create a new set of initial advice containing only 48 rules (see Section 5.3 for more details). I marked the non-terminal representing these rules with the † symbol to distinguish them from the other rules.

†home_page_rules_M →

```
WHEN ( insideMetaWords() AND
          ( consecutive( “home page” ) OR
            consecutive( anyOf( “homepage” “home-page” ) ) OR
            consecutive( “personal” anyOf( “info” “information” ) ) ) )
THEN suggest showing page |

WHEN consecutive( “curriculum” anyOf( “vitae” “vita” ) )
THEN weakly suggest doing both |

WHEN consecutiveInHypertext( “curriculum” anyOf(“vitae” “vita”) )
THEN suggest doing both |

WHEN consecutiveInHypertext( “my” anyOf(“vitae” “vita”) )
THEN suggest following link |

WHEN consecutiveInHypertext( “my” anyOf(“cv” “resume”) )
THEN suggest following link |

WHEN consecutive( “my” anyOf( “resume” “cv” “vita” “vitae” ) )
THEN suggest both |

WHEN consecutive( “my” anyOf( “homepage” “home” ) )
THEN suggest both |

WHEN consecutiveInaSection(personal anyOf(“info” “information”))
THEN weakly suggest doing both |
```

```
WHEN NofM( 2,
          anywhereInSections( “personal” )
          anywhereInSections( “information” )
          anywhereInSections( “info” )
          anywhereInSections( “projects” )
          anywhereInSections( “interests” ) )
THEN weakly suggest doing both |

ScaleLinearlyBy(
  consecutive(anyOf( ?LastName ?MiscWord1 ?MiscWord2 ?MiscWord3) ),
  ( anywhereInWindow(“email”) OR anywhereInWindow(“e-mail”)
    OR anywhereInWindow(“mailto”) ),
  ( anywhereInWindow(“phone”) OR anywhereInWindow(“555-1234”)
    OR anywhereInWindow(“fax”) OR anywhereInWindow(“telephone”) ),
  ( anywhereInWindow(“department”) OR anywhereInWindow(“work”)
    OR anywhereInWindow(“office”) OR anywhereInWindow(“dept”) ),
  ( anywhereInWindow(“address”) OR anywhereInWindow(“mailing”) ) ) )
THEN strongly suggest doing both |

ScaleLinearlyBy(
  consecutive(anyOf( ?LastName ?MiscWord1 ?MiscWord2 ?MiscWord3) ),
  ( anywhereOnPage(“email”) OR anywhereOnPage(“e-mail”)
    OR anywhereOnPage(“mailto”) ),
  ( anywhereOnPage(“phone”) OR anywhereOnPage(“fax”)
    OR anywhereOnPage(“555-1234”) OR anywhereOnPage(“telephone”) ),
  ( anywhereOnPage(“department”) OR anywhereOnPage(“work”)
    OR anywhereOnPage(“office”) OR anywhereOnPage(“dept”) ),
  ( anywhereOnPage(“address”) OR anywhereOnPage(“mailing”) ) ) )
THEN suggest doing both |

WHEN consecutive(anyOf(“research” “recent”)
                 anyOf(“summary” “publications”) )
THEN weakly suggest doing both |

WHEN consecutive( “recent publications” )
THEN weakly suggest doing both |

WHEN ( insideEmailAddress() AND
       consecutive( ?UrlHostWord1 ?UrlHostWord2
                    ?UrlHostWord3 ) )
THEN suggest both |
```

WHEN (insideEmailAddress() AND
consecutive(*?UrlHostWord2 ?UrlHostWord3*))
THEN suggest both |

WHEN consecutiveInURL(*?UrlHostWord1 ?UrlHostWord2*
?UrlHostWord3)
THEN suggest showing page |

WHEN consecutiveInUrl(*?UrlHostWord2 ?UrlHostWord3*)
THEN suggest showing page |

WHEN consecutiveInHyperlink(*?UrlHostWord1 ?UrlHostWord2*
?UrlHostWord3)
THEN suggest following link |

WHEN consecutiveInHyperlink(*?UrlHostWord2 ?UrlHostWord3*)
THEN suggest following link |

ScaleLinearlyBy(
anywhereOnPage("bio"),
anywhereOnPage("interests"),
anywhereOnPage("hobbies"),
anywhereOnPage("resume"),
anywhereOnPage("cv"),
anywhereOnPage("vita"),
anywhereOnPage("vitae"),
anywhereOnPage("degrees"),
anywhereOnPage("employment"),
anywhereOnPage("office"),
anywhereOnPage("courses"),
anywhereOnPage("classes"),
anywhereOnPage("education"),
anywhereOnPage("dept"))
THEN strongly suggest showing page |

```
ScaleLinearlyBy(
   anywhereInWindow(“bio”),
   anywhereInWindow(“interests”),
   anywhereInWindow(“hobbies”),
   anywhereInWindow(“resume”),
   anywhereInWindow(“cv”),
   anywhereInWindow(“vita”),
   anywhereInWindow(“vitae”),
   anywhereInWindow(“degrees”),
   anywhereInWindow(“employment”),
   anywhereInWindow(“office”),
   anywhereInWindow(“courses”),
   anywhereInWindow(“classes”),
   anywhereInWindow(“education”),
   anywhereInWindow(“dept”) )
THEN strongly suggest following link |

WHEN ( anywhereInWindow(“links”) AND
          consecutive(anyOf(“interests” “interesting” “cool”)) )
THEN suggest showing page |

WHEN ( anywhereInWindow(“links”) AND
          consecutive(anyOf(“recommended” “stuff”)) )
THEN suggest showing page |

WHEN consecutiveInaSection(anyOf(“topics” “areas”) “of interest”)
THEN suggest doing both |

WHEN consecutiveInTitle( “main page” )
THEN weakly suggest doing both |

WHEN ( consecutive( “contact information” ) AND
          anywhereOnPage( ?LastName ) )
THEN strongly do both |

WHEN consecutive( “check your spelling” )
THEN strongly avoid both |

WHEN consecutive( “search tips” )
THEN strongly avoid both
```

The union of the rules in the non-terminal tokens **home_page_rules_A** through **home_page_rules_M** create the 76 advice rules used in most of the home-page finder experiments. The union of the rules in **home_page_rules_A** through **home_page_rules_L** create the 48 advice rules used in some of the 1998 home-page finder experiments.

Appendix C

Advice Used in the Seminar-Announcement Extractor

This appendix presents the advice rules for the speaker and location slots of the seminar-announcement extractor agent. Recall that the function named *consecutive* takes a sequence of words and returns true if they appear as a phrase on the page. Otherwise, it returns false.

To extract the speaker name, I used the following four variables:

1. `?FirstName` ← First name or initial of a person
2. `?NickName` ← Nickname of a person
3. `?MiddleName` ← Middle name or initial of a person
4. `?LastName` ← Last name of a person

The advice rules used for the speaker slot in Backus-Naur form (Aho, Sethi, and Ullman 1986) are listed below. The non-terminal tokens **talk_VB** and **talk_VBG** used in the rules refer to verbs in base form and present participle form, respectively. Recall that I choose not to do stemming of words in this study.

spk_rules →

WHEN consecutive(**title spk_name**)
THEN strongly suggest showing page |

WHEN consecutive(**spk_name** , **degree**)
THEN strongly suggest showing page |

WHEN consecutive(**spk_intro . spk_name**)
THEN strongly suggest showing page |

WHEN consecutive(**spk_name** "'s" **talk_noun**)
THEN strongly suggest showing page |

WHEN consecutive(**spk_name** "will" **talk_VB**)
THEN strongly suggest showing page |

WHEN consecutive(**spk_name** "will be" **talk_VBG**)
THEN strongly suggest showing page |

WHEN consecutive("presented by" **spk_name**)
THEN strongly suggest showing page |

WHEN consecutive("talk by" **spk_name**)
THEN strongly suggest showing page |

WHEN **spk_name** THEN weakly suggest showing page

spk_name →
?LastName/NNP |
?FirstName/NNP *?LastName*/NNP |
?NickName/NNP *?LastName*/NNP |
?FirstName/NNP *?MiddleName*/NNP *?LastName*/NNP |
?NickName/NNP *?MiddleName*/NNP *?LastName*/NNP

title → "mr" | "ms" | "mrs" | "dr" | "prof" | "professor" | "mr." | "ms." | "mrs." | "dr." | "prof."

degree → "ba" | "bs" | "ms" | "ma" | "jd" | "md" | "phd" | "b.a." | "b.s." | "m.s." | "m.a." | "j.d." | "m.d." | "ph.d."

spk_intro → "visitor" | "who" | "seminar" | "lecturer" | "colloquium" | "speaker" | "talk"

talk_noun → "talk" | "presentation" | "lecture" | "speech"

talk_VB → "talk" | "lecture" | "speak" | "present"

talk_VBG → "talking" | "lecturing" | "speaking" | "presenting"

To extract the location name, I used the following four variables:

1. *?LocNumber* ← A cardinal number representing a room number, a building number, etc
2. *?LocNameA* ← First word in the name of a building, a street, etc
3. *?LocNameB* ← Second word in the name of a building, a street, etc
4. *?LocNameC* ← Third word in the name of a building, a street, etc

The advice rules used for the location slot in Backus-Naur form (Aho, Sethi, and Ullman 1986) are:

loc_rules →

WHEN consecutive(**loc_name_tagged**)
THEN strongly suggest showing page |

WHEN consecutive(**loc_name**)
THEN weakly suggest showing page |

WHEN consecutive(**loc_name_tagged loc_tokens**)
THEN strongly suggest showing page |

WHEN consecutive("in" **loc_name loc_tokens**)
THEN strongly suggest showing page |

WHEN consecutive(**loc_tokens loc_name_tagged**)
THEN strongly suggest showing page |

WHEN consecutive(**loc_tokens loc_name**)
THEN suggest showing page |

WHEN consecutive(**loc_tokens . loc_name_tagged**)
THEN strongly suggest showing page |

WHEN consecutive(**loc_tokens . loc_name**)
THEN suggest showing page |

WHEN consecutive(**loc_intro . loc_name_tagged**)
THEN strongly suggest showing page |

WHEN consecutive(**loc_intro . loc_name**)
THEN suggest showing page

loc_name_tagged →
?LocNumber/CD |
?LocNameA/NNP |
?LocNumber/CD *?LocNameA*/NNP |
?LocNameA/NNP *?LocNumber*/CD |
?LocNameA/NNP *?LocNameB*/NNP *?LocNumber*/CD |
?LocNumber/CD *?LocNameA*/NNP *?LocNameB*/NNP *?LocNameC*/NNP|
?LocNameA/NNP *?LocNameB*/NNP *?LocNameC*/NNP *?LocNumber*/CD

loc_name →
?LocNumber |
?LocNameA |
?LocNumber ?LocNameA |
?LocNameA ?LocNumber |
?LocNumber ?LocNameA ?LocNameB |
?LocNameA ?LocNameB ?LocNumber |
?LocNumber ?LocNameA ?LocNameB ?LocNameC |
?LocNameA ?LocNameB ?LocNameC ?LocNumber

loc_tokens → "hall" | "auditorium" | "building" | "bldg" | "center" |
"campus" | "school" | "university" | "conference" | "conf" |
"room" | "rm" | "floor" | "inst" | "institute" | "wing" |
"union" | "college" | "office" | "lounge" | "lab" | "laboratory" |
"library"| "classroom" | "tower" | "street" | "avenue" |
"alley" | "road" | "drive"| "circle" | "trail" | "st" | "ave" |
"rd" | "dr" | "cr" | "tr"

loc_intro → "place" | "where" | "location"

Appendix D

Advice Used in the Subcellular-Localization Extractor

This appendix presents two sets of rules for the subcellular-localization extractor. The first set of rules were written by Michael Waddell, who is an M.D./Ph.D. student at University of Wisconsin-Madison. The second set of rules were written by me. Experiments on the second set of advice rules are reported in Eliassi-Rad and Shavlik (2001b).

D.1 Michael Waddell's Rules

As mentioned above, the rules in this section were written by Michael Waddell, who is an M.D./Ph.D. student at University of Wisconsin-Madison. When writing these rules, Mr. Waddell focused only on how to teach the task to another person who could read basic English, but was unfamiliar with the field of biochemistry and its terminology.

Recall that the function named *consecutive* takes a sequence of words and returns true if they appear as a phrase on the page. Otherwise, it returns false. Also, $\cdot$ is one of WAWA's wild card tokens and represents any single word or punctuation. WAWA's other wild card token is $*$, which represents zero of more words or punctuations.

For this domain, I removed the stop words and stemmed the remaining words. The advice rules in Backus-Naur form (Aho, Sethi, and Ullman 1986) are listed below.

protein_location_rules_A $\rightarrow$

WHEN nounPhrase(*?ProteinName*)
THEN strongly suggest show page |

WHEN nounPhrase(*?LocationName*]
THEN strongly suggest show page

protein_location_rules_B $\rightarrow$

WHEN (**protein_name**/unknownWord or **protein_name**/cardinalNumber)
THEN strongly suggest showing page |

WHEN (consecutive(**protein_name protein_associates**) OR
consecutive(**protein_associates protein_name**))
THEN strongly suggest showing page |

WHEN consecutive(**location_marker** $*$ **location_name**)
THEN suggest showing page

protein_location_rules_C $\rightarrow$

WHEN consecutive(**protein_name** "in" **location_name**)
THEN strongly suggest showing page |

WHEN consecutive(**protein_name** $*$ **location_marker**)
THEN suggest showing page |

WHEN consecutive(**protein_name** $*$ **location_marker** $*$ **location_name**)
THEN strongly suggest showing page |

WHEN (consecutive(**protein_name** $*$ **negatives** $*$ **location_marker**) OR
consecutive(**protein_name** $*$ **location_marker** $*$ **negatives**))
THEN avoid showing page |

WHEN (consecutive(**location_marker** $*$ **negatives** $*$ **location_name**) OR
consecutive(**negatives** $*$ **location_marker** $*$ **location_name**))
THEN avoid showing page |

WHEN (consecutive(**protein_name** $*$ **negatives** $*$
location_marker $*$ **location_name**) OR
consecutive(**protein_name** $*$ **location_marker** $*$
negatives $*$ **location_name**))
THEN strongly avoid showing page |

WHEN consecutive(**protein_name** $*$ **passive_voice location_name**)
THEN suggest showing page

protein_name $\rightarrow$ *?ProteinName*/NounPhrase

location_name $\rightarrow$ *?LocationName*/NounPhrase

protein_associates $\rightarrow$ "protein" |"enzyme" | "mutant" | "gene"

locations_words $\rightarrow$ "accompany" | "accumulate" | "adhere" | "anchor" |
"associate" | "attach" | "be" | "bind" | "coexist" |
"cofractionate" | "confine" | "connect" | "contain" | "couple" |
"deposit" | "discover" | "embed" | "encase" | "enclose" |
"establish" | "exist" | "export" | "find" | "fix" | "imbed" |
"incase" | "inclose"| "infiltrate" | "infuse" | "inhabit" |
"join" | "juxtapose" | "lie" | "localize" | "locate" |
"lodge" | "notice" | "perch" | "place" | "plant" | "posit" |
"position" | "put" | "remain" | "reposit" | "root" | "seat" |
"see" | "set" | "settle" | "situate" | "station" | "surround" |
"touch"

location_noun $\rightarrow$ "association" | "coexistence" | "contact" | "localization" |
"location" | "presence"

location_marker →

    location_words/Verb |
    location_words/Adjective |
    location_noun

negatives → "not" | "do not" | "no" | "don 't"

passive_voice → ·/pastTenseVerb | ·/pastParticipleVerb

In Section 7.4, Figure 27 refers to three different groups of advice rules. The first group, called *Group A*, only included the advice rules listed in **protein_location_rules_A**. The second group, called *Group B*, contained the union of the advice rules in **protein_location_rules_A** and **protein_location_rules_B**. Finally, the third group, called *Group C*, had the union of the advice rules in **protein_location_rules_A**, **protein_location_rules_B**, and **protein_location_rules_C**.

The following advice rules indicate properties of proteins and their locations. These rules were not compiled into the network. Instead, they were used as special words in the *prior mean score* for the stochastic selector (see Section 6.3).

- Common locations within a cell are "nucleus,i," "membrane(s)," "golgi," "vesicle(s)," "cytoskeleton(s)," "mitochondrion,a," "endoplasmic reticulum,i," "cytoplasm," "vacuole(s)," "cell wall(s)."
- Terms "locus," "chromosome," or "band" are found within any of the protein or location phrases.

D.2 My Rules

I wrote the following rules for the subcellular-localization domain. Experiments on these advice rules are reported in Eliassi-Rad and Shavlik (2001b). Please note that except for the rules referring to the terminals in **protein_associates**, the rest of the rules can be used in any task that is about locating an object.

protein_location_rules →

WHEN nounPhrase(*?ProteinName*)
THEN strongly suggest show page |

WHEN nounPhrase(*?LocationName*]
THEN strongly suggest show page |

WHEN consecutive(**protein_name protein_associates**)
THEN strongly suggest showing page |

WHEN consecutive(**protein_name** ·/VerbPhrase **location_name**)
THEN suggest showing page |

WHEN consecutive(**protein_name**
·/VerbPhrase ·/PrepositionalPhrase
location_name)
THEN suggest showing page |

WHEN consecutive(**protein_name** ·/VerbPhrase)
THEN weakly suggest showing page |

WHEN consecutive(·/VerbPhrase **location_name**)
THEN weakly suggest showing page |

WHEN consecutive(**protein_name** "at" **location_name**)
THEN suggest showing page |

WHEN consecutive(· "in" **location_name** "of" **protein_name**)
THEN suggest showing page |

WHEN consecutive(**protein_name** "and" ·/NounPhrase
·/VerbPhrase **location_name**)
THEN suggest showing page |

WHEN consecutive(**protein_name** "and" ·/NounPhrase
"at" **location_name**)
THEN suggest showing page |

WHEN consecutive(**protein_name** “and” $\cdot$/NounPhrase
“in” **location_name**)
THEN suggest showing page

protein_name $\rightarrow$ *?ProteinName*/NounPhrase

location_name $\rightarrow$ *?LocationName*/NounPhrase

protein_associates $\rightarrow$ “protein” | “mutant” | “gene”

Appendix E

Advice Used in the Disorder-Association Extractor

This appendix presents the advice rules for the disorder-association extractor agent. These rules were written by Michael Waddell, who is an M.D./Ph.D. student at University of Wisconsin-Madison. When writing these rules, Mr. Waddell focused only on how to teach the task to another person who could read basic English, but was unfamiliar with the field of biochemistry and its terminology.

Recall that the function named *consecutive* takes a sequence of words and returns true if they appear as a phrase on the page. Otherwise, it returns false. Also, $\cdot$ is one of WAWA's wild card tokens and represents any single word or punctuation. WAWA's other wild card token is $*$, which represents zero of more words or punctuations.

For this domain, I removed the stop words and stemmed the remaining words. The advice rules in Backus-Naur form (Aho, Sethi, and Ullman 1986) are listed below.

gene_disease_rules_A $\rightarrow$

WHEN nounPhrase(*?GeneName*)
THEN strongly suggest show page |

WHEN nounPhrase(*?DiseaseName*)
THEN strongly suggest show page

gene_disease_rules_B →

WHEN (**gene_name**/unknownWord or **gene_name**/cardinalNumber)
THEN strongly suggest showing page |

WHEN consecutive(**gene_name gene_trailers**)
THEN strongly suggest showing page |

WHEN (consecutive(**disease_name disease_associates**) OR
consecutive(**disease_associates disease_name**))
THEN strongly suggest showing page |

WHEN consecutive(**disease_name disease_trailers**)
THEN strongly suggest showing page |

WHEN (consecutive(**verb_or_adj** ∗ **gene_name**) OR
consecutive(**gene_name** ∗ **verb_or_adj**))
THEN suggest showing page |

WHEN (consecutive(**verb_or_adj** ∗ **disease_name**) OR
consecutive(**disease_name** ∗ **verb_or_adj**))
THEN suggest showing page

gene_disease_rules_C →

WHEN (consecutive(**gene_name** , **disease_name**) OR
consecutive(**disease_name** , **gene_name**))
THEN strongly suggest showing page |

WHEN (consecutive(**gene_name** ∗ **verb_or_adj** ∗ **disease_name**) OR
consecutive(**disease_name** ∗ **verb_or_adj** ∗ **gene_name**))
THEN strongly suggest showing page |

WHEN (consecutive(**verb_or_adj** ∗ **negatives** ∗ **gene_name**) OR
consecutive(**gene_name** ∗ **negatives** ∗ **verb_or_adj**))
THEN avoid showing page |

WHEN (consecutive(**verb_or_adj** $*$ **negatives** $*$ **disease_name**) OR
consecutive(**disease_name** $*$ **negatives** $*$ **verb_or_adj**))
THEN avoid showing page |

WHEN (consecutive(**gene_name** $*$ **verb_or_adj negatives** $*$ **disease_name**)
OR consecutive(**gene_name** $*$ **negatives verb_or_adj** $*$ **disease_name**)
OR consecutive(**disease_name** $*$ **negatives verb_or_adj** $*$ **gene_name**)
OR consecutive(**disease_name** $*$ **verb_or_adj negatives** $*$ **gene_name**))
THEN strongly avoid showing page |

WHEN consecutive(**gene_name** $*$ **passive_voice** $*$ **disease_name**)
THEN suggest showing page

gene_name $\rightarrow$ *?GeneName*/NounPhrase

disease_name $\rightarrow$ *?DiseaseName*/NounPhrase

gene_trailers $\rightarrow$ "deletion" | "inversion" | "mutation" | "carrier" | "transporter" |
"receptor" | "gene" | "locus" | "loci" | "variant" | "allele"

gene_words $\rightarrow$ "present" | "isolate" | "associate" | "mediate" | "link" |
"mutate" | "complement" | "predispose" | "cause" | "lead" |
"explain" | "increase" | "produce" | "result"

disease_associates $\rightarrow$ "cancer" | "tumor" | "tumour" | "dysplasia" | "syndrome" |
"disorder" | "disease" | "dystrophy" | "deficiency" |
"familial" | "human"

disease_trailers $\rightarrow$ "patient" | "family"

verb_or_adj $\rightarrow$ **gene_words**/Verb | **gene_words**/Adjectives

negatives $\rightarrow$ "not" | "do not" | "no" | "don 't"

passive_voice $\rightarrow$ $\cdot$/pastTenseVerb | $\cdot$/pastParticipleVerb

In Section 7.4, Figure 29 refers to three different groups of advice rules. The first group, called *Group A*, only included the advice rules listed in **gene_disease_rules_A**. The second group, called *Group B*, contained the union of the advice rules in **gene_disease_rules_A** and **gene_disease_rules_B**. Finally, the third group, called *Group C*, had the union of the advice rules in **gene_disease_rules_A**, **gene_disease_rules_B**, and **gene_disease_rules_C**.

The following advice rules indicate properties of genes and their genetic disorders. These rules were not compiled into the network. Instead, they were used as special words in the *prior mean score* for the stochastic selector (see Section 6.3).

- Sometimes the gene name and the disease name are the same.
- Disease names commonly have one of the following suffixes: "-oma," "-uria," "-emia," "-ism," "-opathy," "-osis," and "-itis."
- A common type of disease is a syndrome. That is,
  disease_name →
  "**gene_name** syndrome" | "**gene_name** syndromes"
- A common type of disease is a deficiency in a gene. That is,
  disease_name →
  "**gene_name** deficiency" | "**gene_name** deficiencies" |
  "deficiency in **gene_name**" | "deficiencies in **gene_name**"

Bibliography

Y. Abu-Mostafa (1995). Hints. *Neural Computation 7*, 639–671.

A. Aho, R. Sethi, and J. Ullman (1986). *Compilers, Principles, Techniques and Tools*. Reading, MA: Addison Wesley.

M. Balabanovic and Y. Shoham (1997). Fab: Content–based, collaborative recommendation. *Communications of ACM 40*, 66–72.

S. Baluja, V. Mittal, and R. Sukthankar (1999). Applying machine learning for high performance named-entity extraction. In *Proceedings of Pacific Association for Computational Linguistics*, Ontario, Canada. Blackwell.

R. K. Belew (2000). *Finding Out About: A Cognitive Perspective on Search Engine Technology and the WWW*. New York, NY: Cambridge University Press.

D. Bikel, R. Schwartz, and R. Weischedel (1999). An algorithm that learns what's in a name. *Machine Learning: Special Issue on Natural Language Learning 34*(1/3), 211–231.

M. Botta and R. Piola (1999). Refining numerical constants in first order logic theories. *Machine Learning 38*, 109–131.

J. Boyan and A. Moore (1998). Learning evaluation functions for global optimization and boolean satisfiability. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, Madison, WI, 3–10. AAAI Press.

J. Boyan and A. Moore (2000). Learning evaluation functions to improve optimization by local search. *Journal of Machine Learning Research 1*, 77–112.

E. Brill (1994). Some advances in rule-based part of speech tagging. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, Seattle, WA, 722–727. AAAI Press.

S. Brin and L. Page (1998). The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems 30*, 107–117.

M. E. Califf (1998). *Relational Learning Techniques for Natural Language Information Extraction*. Ph. D. Thesis, Department of Computer Sciences, University of Texas, Austin, TX.

M. E. Califf and R. Mooney (1999). Relational learning of pattern-match rules for information extraction. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, Orlando, FL, 328–334. AAAI Press.

E. Coffman, M. Garey, and D. Johnson (1996). Approximation Algorithms for Bin Packing: A Survey. In D. Hochbaum (Ed.), *Approximation Algorithms for NP-Hard Problems*. PWS Publishing.

M. Craven and J. Kumlien (1999). Constructing biological knowledge-bases by extracting information from text sources. In *Proceedings of the Seventh International Conference on Intelligent Systems for Molecular Biology*, Heidelberg, Germany, 77–86. AAAI Press.

M. W. Craven and J. W. Shavlik (1992). Visualizing learning and computation in artificial neural networks. *International Journal on Artificial Intelligence Tools 1*(3), 399–425.

M. W. Craven and J. W. Shavlik (1996). Extracting tree-structured representations of trained networks. In *Advances in Neural Information Processing Systems*, Volume 8, Denver, CO, 24–30. MIT Press.

N. Cristianini and J. Shawe-Taylor (2000). *An Introduction to Support Vector*

Machines and Other Kernel-Based Learning Methods. Cambridge University Press.

W. Croft, H. Turtle, and D. Lewis (1991). The use of phrases and structured queries in information retrieval. In *Proceedings of the Fourteenth International ACM SIGIR Conference on R & D in Information Retrieval*, Chicago, IL, 32–45. ACM Press.

A. Dempster, N. Laird, and D. Rubin (1977). Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society 39*(1), 1–38.

J. Diederich (1989). Learning by instruction in connectionist systems. In *Proceedings of the Sixth International Workshop on Machine Learning*, Ithaca, NY, 66–68. Morgan Kaufmann.

C. Drummond, D. Ionescu, and R. Holte (1995). A learning agent that assists the browsing of software libraries. Technical Report TR-95-12, University of Ottawa, Ottawa, Canada.

T. Eliassi-Rad and J. Shavlik (2001a). A system for building intelligent agents that learn to retrieve and extract information. *International Journal on User Modeling and User-Adapted Interaction, Special Issue on User Modeling and Intelligent Agents*. (To appear).

T. Eliassi-Rad and J. Shavlik (2001b). A theory-refinement approach to information extraction. In *Proceedings of the Eighteenth International Conference on Machine Learning*, Williamstown, MA, 130–137. Morgan Kaufmann.

J. Elman (1991). Distributed representations, simple recurrent networks, and grammatical structure. *Machine Learning 7*, 195–225.

O. Etzioni and D. Weld (1995). Intelligent agents on the internet: Fact, fiction, and forecast. *IEEE Expert 10*, 44–49.

W. B. Frakes and R. Baeza-Yates (1992). *Information Retrieval: Data Structures & Algorithms*. Prentice Hall.

D. Freitag (1998a). Information extraction from HTML: Application of a general machine learning approach. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, Madison, WI, 517–523. AAAI Press.

D. Freitag (1998b). *Machine Learning for Information Extraction in Informal Domains*. Ph. D. Thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA.

D. Freitag (1998c). Relational learning of pattern-match rules for information extraction. In *Proceedings of the Fifteenth International Conference on Machine Learning*, Madison, WI, 161–169. Morgan Kaufmann.

D. Freitag and N. Kushmerick (2000). Boosted wrapper induction. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, Austin, TX, 577–583. AAAI Press.

D. Freitag and A. McCallum (1999). Information extraction with HMMs and shrinkage. In *Notes of the Sixteenth National Conference on Artificial Intelligence Workshop on Machine Learning for Information Extraction*, Orlando, FL, 31–36. AAAI Press.

D. Freitag and A. McCallum (2000). Information extraction with HMM structures learned by stochastic optimization. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, Austin, TX, 584–589. AAAI Press.

J. Goecks and J. Shavlik (2000). Learning users' interests by unobtrusively observing their normal behavior. In *Proceedings of the 2000 International Conference on Intelligent User Interfaces*, New Orleans, LA, 129–132. ACM Press.

D. Gordon and D. Subramanian (1994). A multistrategy learning scheme for agent knowledge acquisition. *Informatica 17*, 331–346.

T. Joachims, D. Freitag, and T. Mitchell (1997). WebWatcher: A tour guide for the World Wide Web. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, Nagoya, Japan, 770–775. AAAI Press.

N. Kushmerick (2000). Wrapper induction: Efficiency and expressiveness. *Artificial Intelligence 118*, 15–68.

S. Lawrence and C. L. Giles (1999). Accessibility of information on the web. *Nature 400*(6740), 107–109.

T. Leek (1997). Information extraction using hidden markov models. Master's Thesis, Department of Computer Science & Engineering, University of California, San Diego.

W. Lehnert (2000). *Information Extraction.* http://www-nlp.cs.umass.edu/nlpie.html.

H. Lieberman (1995). Letzia: An agent that assists web browsing. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, Montreal, Canada, 924–929. AAAI Press.

R. Maclin (1995). *Learning from Instruction and Experience: Methods for Incorporating Procedural Domain Theories into Knowledge-Based Neural Networks.* Ph. D. Thesis, Department of Computer Sciences, University of Wisconsin, Madison, WI. (Also appears as UW Technical Report CS-TR-95-1285).

R. Maclin and J. Shavlik (1996). Creating advice-taking reinforcement learners. *Machine Learning 22*, 251–281.

A. McCallum and K. Nigam (1998). A comparison of event models for naive Bayes text classification. In *Notes of the Fifteenth National Conference on*

Artificial Intelligence Workshop on Learning for Text Categorization, Madison, WI, 41–48. AAAI Press.

A. McCallum, K. Nigam, J. Rennie, and K. Seymore (1999). Building domain-specific search engines with machine learning techniques. In *American Association for Artificial Intelligence Spring 1999 Symposium*, Stanford University, CA, 28–39. AAAI Press.

A. McCallum, K. Nigam, J. Rennie, and K. Seymore (2000). Automating the construction of internet portals with machine learning. *Information Retrieval Journal 3*, 127–163.

A. McCallum, R. Rosenfeld, T. Mitchell, and A. Ng (1998). Improving text classification by shrinkage in a hierarchy of classes. In *Proceedings of the Fifteenth International Conference on Machine Learning*, Madison, WI, 359–367. Morgan Kaufmann.

G. Miller (1995). WordNet: A lexical database for English. *Communications of the ACM 38*, 39–41.

T. Mitchell (1997). *Machine Learning*. McGraw-Hill.

S. Muggleton (1995). *Foundations of inductive logic programming*. Englewood Cliffs, NJ: Prentice Hall.

NLM (2001). *National Library of Medicine's MEDLINE Database*. http://www.ncbi.nlm.nih.gov/PubMed/.

D. Ourston and R. Mooney (1994). Theory refinement: Combining analytical and empirical methods. *Artificial Intelligence 66*, 273–309.

M. Pazzani and D. Kibler (1992). The utility of knowledge in inductive learning. *Machine Learning 9*, 57–94.

M. Pazzani, J. Muramatsu, and D. Billsus (1996). Syskill & Webert: Identifying interesting web sites. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Portland, OR, 54–61. AAAI Press.

M. F. Porter (1980). An algorithm for suffix stripping. *Program 14*(3), 130–137.

J. Quinlan (1990). Learning logical definitions from relations. *Machine Learning 5*, 239–266.

L. Rabiner and B. Juang (1993). *Fundamentals of Speech Recognition.* Englewood Cliffs, NJ: Prentice-Hall.

S. Ray and M. Craven (2001). Representing sentence structure in hidden markov models for information extraction. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, Seattle, WA, 1273–1279. AAAI Press.

J. Rennie and A. McCallum (1999). Using reinforcement learning to spider the web efficiently. In *Proceedings of the Sixteenth International Conference on Machine Learning*, Bled, Slovenia, 335–343. Morgan Kaufmann.

E. Rich and K. Knight (1991). *Artificial Intelligence.* McGraw Hill.

E. Riloff (1996). Using learned extraction patterns for text classification. In S. Wermter, E. Riloff, and G. Scheler (Eds.), *Connectionist, Statistical, and Symbolic Approaches to Learning for Natural Language Processing*, 275–289. Springer-Verlag.

E. Riloff (1998). *The Sundance Sentence Analyzer.* http://www.cs.utah.edu/projects/nlp/.

E. Riloff and W. Lehnert (1994). Information extraction as a basis for high-precision text classification. *ACM Transactions on Information Systems 12*(3), 296–333.

D. E. Rose (1994). *A Symbolic and Connectionist Approach to Legal Information Retrieval*. Lawrence Erlbaum Associates.

D. Rumelhart, G. Hinton, and R. Williams (1986). Learning internal representations by error propagation. In D. Rumelhart and J. McClelland (Eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Volume 1, 318–363. MIT Press.

D. Rumelhart and J. McClelland (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. MIT Press.

S. Russell and P. Norvig (1995). *Artificial Intelligence: A Modern Approach*. Prentice Hall.

G. Salton (1991). Developments in automatic text retrieval. *Science 253*, 974–979.

G. Salton and C. Buckley (1988). Term-weighting approaches in automatic text retrieval. *Information Processing and Management 24*(5), 513–523.

R. Schapire and Y. Singer (1998). Improved boosting algorithms using confidence-rated predictions. In *Proceedings of the Eleventh Annual Conference on Computational Learning Theory*, Madison, WI, 80–91. ACM Press.

T. Sejnowski and C. Rosenberg (1987). Parallel networks that learn to pronounce English text. *Complex Systems 1*, 145–168.

B. Selman, H. Kautz, and B. Cohen (1996). Local search strategies for satisfiability testing. *DIMACS Series in Discrete Mathematics and Theoretical CS 26*, 521–531.

K. Seymore, A. McCallum, and R. Rosenfeld (1999). Learning hidden Markov model structure for information extraction. In *Notes of the Sixteenth National Conference on Artificial Intelligence Workshop on Machine Learning for Information Extraction*, Orlando, FL, 37–42. AAAI Press.

J. Shakes, M. Langheinrich, and O. Etzioni (1997). Dynamic reference sifting: A case study in the homepage domain. In *Proceedings of the Sixth International World Wide Web Conference*, Santa Clara, CA, 189–200.

J. Shavlik, S. Calcari, T. Eliassi-Rad, and J. Solock (1999). An instructable, adaptive interface for discovering and monitoring information on the world-wide web. In *Proceedings of the 1999 International Conference on Intelligent User Interfaces*, Redondo Beach, CA, 157–160. ACM Press.

J. Shavlik and T. Eliassi-Rad (1998a). Building intelligent agents for web-based tasks: A theory-refinement approach. In *Proceedings of the Conference on Automated Learning and Discovery Workshop on Learning from Text and the Web*, Pittsburgh, PA.

J. Shavlik and T. Eliassi-Rad (1998b). Intelligent agents for web-based tasks: An advice-taking approach. In *Notes of the Fifteenth National Conference on Artificial Intelligence Workshop on Learning for Text Categorization*, Madison, WI, 63–70. AAAI Press.

S. Soderland (1997). Learning to extract text-based information from the World Wide Web. In *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining*, Newport Beach, CA, 251–254. AAAI Press.

S. Soderland (1999). Learning information extraction rules for semi-structured and free text. *Machine Learning: Special Issue on Natural Language Learning 34*(1/3), 233–272.

R. Sutton (1988). Learning to predict by the methods of temporal differences. *Machine Learning 3*, 9–44.

R. S. Sutton and A. G. Barto (1998). *Reinforcement Learning*. MIT Press.

G. G. Towell and J. W. Shavlik (1994). Knowledge-based artificial neural networks. *Artificial Intelligence 70*(1/2), 119–165.

C. J. van Rijsbergen (1979). *Information Retrieval* (second ed.). London: Buttersworths.

C. Watkins (1989). *Learning from Delayed Rewards.* Ph. D. Thesis, King's College, Cambridge.

M. Wooldridge and N. Jennings (1995). Intelligent agents: Theory and practice. *Knowledge Engineering Review 10*, 45–52.

H. Zaragoza and P. Gallinari (1998). An automatic surface information extraction system using hierarchical IR and stochastic IE. In *Notes of the Tenth European Conference on Machine Learning Workshop on Text Mining*, Chemnitz, Germany. Springer-Verlag.