# Automating the Construction of Internet Portals with Machine Learning

Andrew Kachites McCallum (mccallum@cs.cmu.edu)
*Just Research and Carnegie Mellon University*

Kamal Nigam (knigam@cs.cmu.edu)
*Carnegie Mellon University*

Jason Rennie (jrennie@ai.mit.edu)
*Massachusetts Institute of Technology*

Kristie Seymore (kseymore@ri.cmu.edu)
*Carnegie Mellon University*

**Abstract.** Domain-specific internet portals are growing in popularity because they gather content from the Web and organize it for easy access, retrieval and search. For example, *www.campsearch.com* allows complex queries by age, location, cost and specialty over summer camps. This functionality is not possible with general, Web-wide search engines. Unfortunately these portals are difficult and time-consuming to maintain. This paper advocates the use of machine learning techniques to greatly automate the creation and maintenance of domain-specific Internet portals. We describe new research in reinforcement learning, information extraction and text classification that enables efficient spidering, the identification of informative text segments, and the population of topic hierarchies. Using these techniques, we have built a demonstration system: a portal for computer science research papers. It already contains over 50,000 papers and is publicly available at *www.cora.justresearch.com*. These techniques are widely applicable to portal creation in other domains.

**Keywords:** spidering, crawling, reinforcement learning, information extraction, hidden Markov models, text classification, naive Bayes, Expectation-Maximization, unlabeled data

## 1. Introduction

As the amount of information on the World Wide Web grows, it becomes increasingly difficult to find just what we want. While general-purpose search engines such as AltaVista and Google offer quite useful coverage, it is often difficult to get high precision, even for detailed queries. When we know that we want information of a certain type, or on a certain topic, a domain-specific Internet portal can be a powerful tool. A *portal* is an information gateway that often includes a search engine plus additional organization and content. Portals are often, though not always, concentrated on a particular topic. They usually offer powerful methods for finding domain-specific information. For example:

- Camp Search (*www.campsearch.com*) allows the user to search for summer camps for children and adults. The user can query and browse the system based on geographic location, cost, duration and other requirements.

- LinuxStart (*www.linuxstart.com*) provides a clearinghouse for Linux resources. It has a hierarchy of topics and a search engine over Linux pages.

- Movie Review Query Engine (*www.mrqe.com*) allows the user to search for reviews of movies. Type a movie title, and it provides links to relevant reviews from newspapers, magazines, and individuals from all over the world.

- Crafts Search (*www.bella-decor.com*) lets the user search web pages about crafts. It also provides search capabilities over classified ads and auctions of crafts, as well as a browseable topic hierarchy.

- Travel-Finder (*www.travel-finder.com*) allows the user to search web pages about travel, with special facilities for searching by activity, category and location.

Performing any of these searches with a traditional, general-purpose search engine would be extremely tedious or impossible. For this reason, portals are becoming increasingly popular. Unfortunately, however, building these portals is often a labor-intensive process, typically requiring significant and ongoing human effort.

This article describes the use of machine learning techniques to automate several aspects of creating and maintaining portals. These techniques allow portals to be created quickly with minimal effort and are suited for re-use across many domains. We present new machine learning methods for spidering in an efficient topic-directed manner, extracting topic-relevant information, and building a browseable topic hierarchy. These approaches are briefly described in the following three paragraphs.

Every search engine or portal must begin with a collection of documents to index. A spider (or crawler) is an agent that traverses the Web, looking for documents to add to the collection. When aiming to populate a domain-specific collection, the spider need not explore the Web indiscriminantly, but should explore in a directed fashion in order to find domain-relevant documents efficiently. We set up the spidering task in a reinforcement learning framework (Kaelbling, Littman, & Moore, 1996), which allows us to precisely and mathematically define optimal behavior. This approach provides guidance for designing an intelligent spider that aims to select hyperlinks optimally. It also indicates how the agent should learn from delayed reward. Our experimental results show that a reinforcement learning spider is twice as efficient in finding domain-relevant documents as a baseline topic-focused spider and three times more efficient than a spider with a breadth-first search strategy.

Extracting characteristic pieces of information from the documents of a domain-specific collection allows the user to search over these features in a way that general search engines cannot. Information extraction, the process of automatically finding certain categories of textual substrings in a document, is well suited to this task. We approach information extraction with a technique from statistical language modeling and speech recognition, namely *hidden Markov models* (Rabiner, 1989). We learn model structure and parameters from a combination of labeled and distantly-labeled data. Our model extracts fifteen different fields from spidered documents with 93% accuracy.

Search engines often provide a hierarchical organization of materials into relevant topics; Yahoo is the prototypical example. Automatically adding documents into a topic hierarchy can be framed as a text classification task. We present extensions to a probabilistic text classifier known as *naive Bayes* (Lewis, 1998; McCallum & Nigam, 1998). The extensions reduce the need for human effort in training the classifier by using just a few keywords per class, a class hierarchy and unlabeled documents in a bootstrapping process. Use of the resulting classifier places documents into a 70-leaf topic hierarchy with 66% accuracy—performance approaching human agreement levels.

The remainder of the paper is organized as follows. We describe the design of an Internet portal built using these techniques in the next section. The following three sections describe the machine learning research introduced above and present their experimental results. We then discuss related work and present conclusions.

## 2. The Cora Portal

We have brought all the above-described machine learning techniques together in a demonstration system: an Internet portal for computer science research papers, which we call "Cora." The system is publicly available at *www.cora.justresearch.com*. Not only does it provide keyword search facilities over 50,000 collected papers, it also places these papers into a computer science topic hierarchy, maps the citation links between papers, provides bibliographic information about each paper,

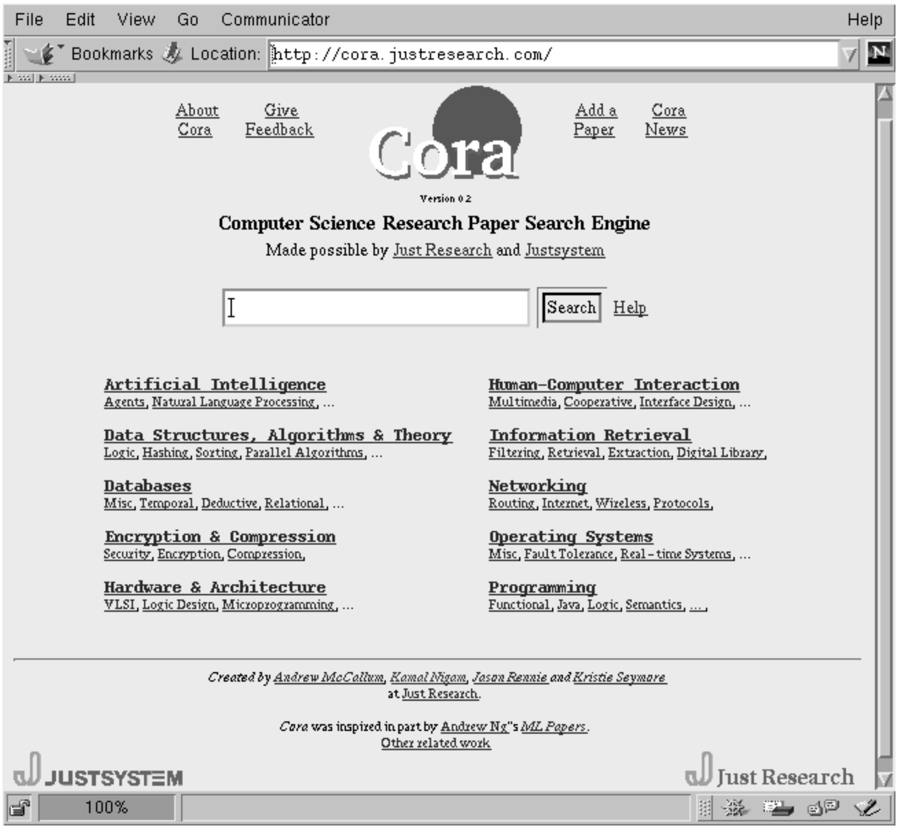



*Figure 1.* A screen shot of the Cora homepage (*www.cora.justresearch.com*). It has a search interface and a hierarchy interface.

and is growing daily. Our hope is that in addition to providing datasets and a platform for testing machine learning research, this search engine will become a valuable tool for other computer scientists, and will complement similar efforts, such as CiteSeer (*www.scienceindex.com*) and the Computing Research Repository (*xxx.lanl.gov/archive/cs*).

We provide three ways for a user to access papers in the repository. The first is through a topic hierarchy, similar to that provided by Yahoo but customized specifically for computer science research. It is available on the homepage of Cora, as shown in Figure 1. This hierarchy was hand-constructed and contains 70 leaves, varying in depth from one to three. Using text classification techniques, each research paper is automatically placed into a topic leaf. The topic hierarchy may be traversed by following hyperlinks from the homepage. Each leaf in the tree contains a list of papers in that research topic. The list can be sorted by the number of references to each paper, or by the degree to

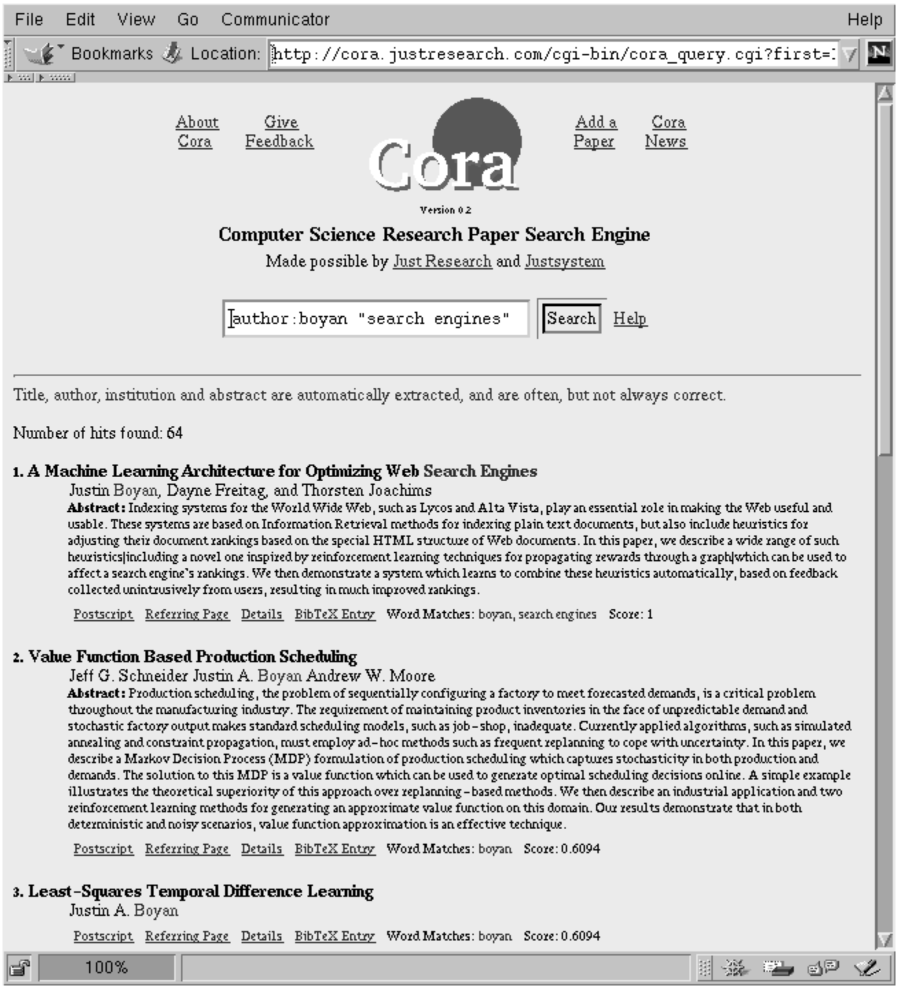
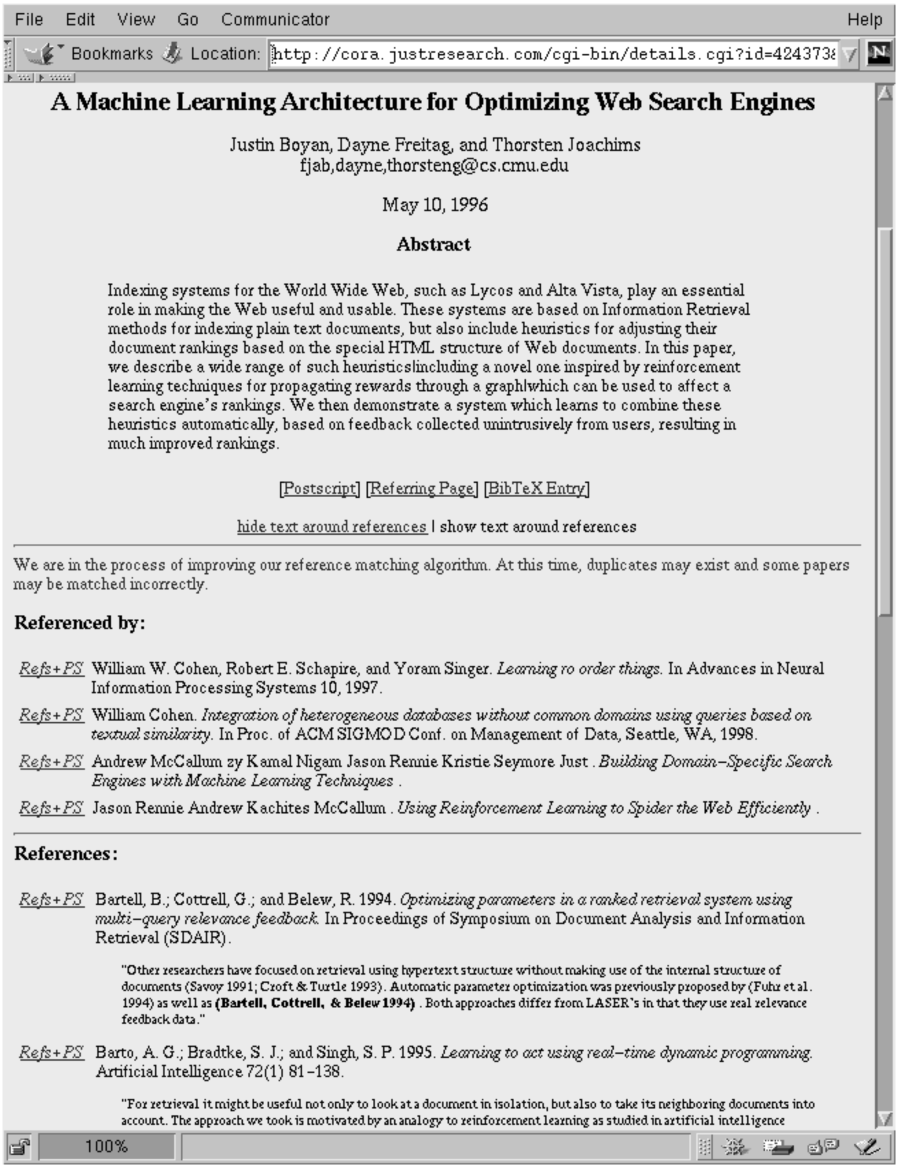



*Figure 2.* A screen shot of the query results page of the Cora search engine. Extracted paper titles, authors and abstracts are provided at this level.

which the paper is a strong "seminal" paper or a good "survey" paper, as measure by the "authority" and "hub" score according to the HITS algorithm (Kleinberg, 1999; Chang, Cohn, & McCallum, 1999).

All papers are indexed into a search engine available through a standard search interface. It supports commonly-used searching syntax for queries, including `+`, `-`, and phrase searching with `""`. It also allows searches restricted to extracted fields, such as authors and titles, as in `author:knuth`. Query response time is usually less than a second. The results of search queries are presented as in Figure 2. While we present no experimental evidence that the ability to restrict search to specific extracted fields improves search performance, it is generally accepted


File Edit View Go Communicator Help

Bookmarks Location: http://cora.justresearch.com/cgi-bin/details.cgi?id=424373

A Machine Learning Architecture for Optimizing Web Search Engines

Justin Boyan, Dayne Freitag, and Thorsten Joachims
fjab,dayne,thorsteng@cs.cmu.edu

May 10, 1996

Abstract

Indexing systems for the World Wide Web, such as Lycos and Alta Vista, play an essential role in making the Web useful and usable. These systems are based on Information Retrieval methods for indexing plain text documents, but also include heuristics for adjusting their document rankings based on the special HTML structure of Web documents. In this paper, we describe a wide range of such heuristicslincluding a novel one inspired by reinforcement learning techniques for propagating rewards through a graphlwhich can be used to affect a search engine's rankings. We then demonstrate a system which learns to combine these heuristics automatically, based on feedback collected unintrusively from users, resulting in much improved rankings.

[Postscript] [Referring Page] [BibTeX Entry]

hide text around references | show text around references

We are in the process of improving our reference matching algorithm. At this time, duplicates may exist and some papers may be matched incorrectly.

Referenced by:

Refs+PS William W. Cohen, Robert E. Schapire, and Yoram Singer. *Learning ro order things.* In Advances in Neural Information Processing Systems 10, 1997.

Refs+PS William Cohen. *Integration of heterogeneous databases without common domains using queries based on textual similarity.* In Proc. of ACM SIGMOD Conf. on Management of Data, Seattle, WA, 1998.

Refs+PS Andrew McCallum zy Kamal Nigam Jason Rennie Kristie Seymore Just . *Building Domain–Specific Search Engines with Machine Learning Techniques* .

Refs+PS Jason Rennie Andrew Kachites McCallum . *Using Reinforcement Learning to Spider the Web Efficiently* .

References:

Refs+PS Bartell, B.; Cottrell, G.; and Belew, R. 1994. *Optimizing parameters in a ranked retrieval system using multi–query relevance feedback.* In Proceedings of Symposium on Document Analysis and Information Retrieval (SDAIR).

"Other researchers have focused on retrieval using hypertext structure without making use of the internal structure of documents (Savoy 1991; Croft & Turtle 1993). Automatic parameter optimization was previously proposed by (Fuhr et al. 1994) as well as **(Bartell, Cottrell, & Belew 1994)** . Both approaches differ from LASER's in that they use real relevance feedback data."

Refs+PS Barto, A. G.; Bradtke, S. J.; and Singh, S. P. 1995. *Learning to act using real–time dynamic programming.* Artificial Intelligence 72(1) 81–138.

"For retrieval it might be useful not only to look at a document in isolation, but also to take its neighboring documents into account. The approach we took is motivated by an analogy to reinforcement learning as studied in artificial intelligence

100%

*Figure 3.* A screen shot of a details page of the Cora search engine. At this level, all extracted information about a paper is displayed, including the citation linking, which are hyperlinks to other details pages.

that such capability increases the users' ability to efficiently find what they want (Bikel, Miller, Schwartz, & Weischedel, 1997).

From both the topic hierarchy and the search results pages, links are provided to "details" pages for individual papers. Each of these pages shows all the relevant information for a single paper, such as title and authors, links to the actual postscript paper, and a citation map that can be traversed either forwards or backwards. One example of this is shown in Figure 3. The citation map allows a user to find details on cited papers, as well as papers that cite the detailed paper. The context of each reference is also provided, giving a brief summary of how the reference is used by the detailed paper. We also provide automatically constructed BibTeX entries, a mechanism for submitting new papers and web sites for spidering, and general Cora information links.

Our web logs show that 40% of the page requests are for searches, 27% for details pages (which show a paper's incoming and outgoing references), 30% are for the topic hierarchy nodes and 3% are for BibTeX entries. The logs show that our visitors use the ability to restrict search to specific extracted fields, but not often; about 3% of queries contain field specifiers; it might have been higher if the front page indicated that this feature were available.

The collection and organization of the research papers for Cora is automated by drawing upon the machine learning techniques described in this paper. The first step of building any portal is the collection of relevant information from the Web. A spider crawls the Web, starting from the home pages of computer science departments and laboratories and looks for research papers. Using reinforcement learning, our spider efficiently explores the Web, following links that are more likely to lead to research papers, and collects all postscript documents it finds.[1] The details of this spidering are described in Section 3. The postscript documents are then converted into plain text by running them through our own modified version of the publicly-available utility *ps2ascii*. If the document can be reliably determined to have the format of a research paper (*i.e.* by matching regular expressions for the headers of an Abstract or Introduction section and a Reference section), it is added to Cora. Using this system, we have found 50,000 computer science research papers, and are continuing to spider for even more.

The beginning of each paper is passed through a learned information extraction system that automatically finds the title, authors, affiliations and other important header information. Additionally, the bibliography section of each paper is located, individual references identified, and each reference automatically broken down into the appropriate fields, such as author, title, journal, and date. This information extraction process is described in Section 4.

Using the extracted information, reference and paper matches are made—grouping citations to the same paper together, and matching

[1] Most computer science papers are in postscript format, though we are adding more formats, such as PDF.

citations to papers in Cora. Of course, many papers that are cited do not appear in the repository. The matching algorithm places a new citation into a group if it's best word-level match is to a citation already in that group, and the match score is above a threshold; otherwise, that citation creates a new group. The word-level match score is determined using the lengths of the citations, and the words occurring in high-content fields (e.g. authors, titles, etc.). This matching procedure is very similar to the Baseline Simple method described by Giles, Bollacker, and Lawrence (1998). Finally, each paper is placed into the computer science hierarchy using a text classification algorithm. This process is described in Section 5.

The search engine is created from the results of the information extraction. Each research paper is represented by the extracted title, author, institution, references, and abstract. Contiguous alphanumeric characters of these segments are converted into word tokens. No stoplists or stemming are used. At query time, result matches are ranked by the weighted log of term frequency, summed over all query terms. The weight is the inverse of the word frequency in the entire corpus. When a phrase is included, it is treated as a single term. No query expansion is performed. Papers are added to the index incrementally, and the indexing time for each document is negligible.

These steps complete the processing of the data necessary to build Cora. The creation of other Internet portals also involves directed spidering, information extraction, and classification. The machine learning techniques described in the following sections are widely applicable to the construction and maintenance of any Internet portal.

## 3. Efficient Spidering

Spiders are agents that explore the hyperlink graph of the Web, often for the purpose of finding documents with which to populate a portal. Extensive spidering is the key to obtaining high coverage by the major Web search engines, such as AltaVista, Google and Lycos. Since the goal of these general-purpose search engines is to provide search capabilities over the Web as a whole, they aim to find as many distinct web pages as possible. Such a goal lends itself to strategies like breadth-first search. If, on the other hand, the task is to populate a domain-specific portal, then an intelligent spider should try to avoid hyperlinks that lead to off-topic areas, and concentrate on links that lead to documents of interest.

In Cora, efficient spidering is a major concern. The majority of the pages in computer science department web sites do not contain links to research papers, but instead are about courses, homework, schedules and admissions information. Avoiding whole branches and neighborhoods of departmental web graphs can significantly improve efficiency and increase the number of research papers found given a finite amount of crawling time. We use reinforcement learning as the setting for efficient spidering in order to provide a formal framework. As in much other work in reinforcement learning, we believe that the best approach to this problem is to formally define the optimal solution that a spider should follow and then to approximate that policy as best as possible. This allows us to understand (1) exactly what has been compromised, and (2) directions for further work that should improve performance.

Several other systems have also studied spidering, but without a framework defining optimal behavior. ARACHNID (Menczer, 1997) maintains a collection of competitive, reproducing and mutating agents for finding information on the Web. Cho, Garcia-Molina, and Page (1998) suggest a number of heuristic ordering metrics for choosing which link to crawl next when searching for certain categories of web pages. Chakrabarti, van der Berg, and Dom (1999) produce a spider to locate documents that are textually similar to a set of training documents. This is called a focused crawler. This spider requires only a handful of relevant example pages, whereas we also require example Web graphs where such relevant pages are likely to be found. However, with this additional training data, our framework explicitly captures knowledge of future reward—the fact that pages leading toward a topic page may have text that is drastically different from the text in topic pages.

Additionally, there are other systems that use reinforcement learning for non-spidering Web tasks. WebWatcher (Joachims, Freitag, & Mitchell, 1997) is a browsing assistant that acts much like a focused crawler, recommending links that direct the user toward a "goal." WebWatcher also uses aspects of reinforcement learning to decide which links to select. However, instead of approximating a $Q$ function for each URL, WebWatcher approximates a $Q$ function for each word and then, for each URL, adds the $Q$ functions that correspond to the URL and the user's interests. In contrast, we approximate a $Q$ function for each URL using regression by classification. LASER (Boyan, Freitag, & Joachims, 1996) is a search engine that uses a reinforcement learning framework to take advantage of the interconnectivity of the Web. It propagates reward values back through the hyperlink graph in order to tune its search engine parameters. In Cora, similar techniques are used to achieve more efficient spidering.

The spidering algorithm we present here is unique in that it represents and takes advantage of future reward—learning features that predict an on-topic document several hyperlink hops away from the current hyperlink. This is particularly important when reward is sparse, or in other words, when on-topic documents are few and far between. Our experimental results bear this out. In a domain with*out* sparse rewards, our reinforcement learning spider that represents future reward performs about the same as a focused spider (both out-perform a breadth-first search spider by three-fold). However, in another domain where reward is more sparse, explicitly representing future reward increases efficiency over a focused spider by a factor of two.

### 3.1. Reinforcement Learning

The term "reinforcement learning" refers to a framework for learning optimal decision making from rewards or punishment (Kaelbling et al., 1996). It differs from supervised learning in that the learner is never told the correct action for a particular state, but is simply told how good or bad the selected action was, expressed in the form of a scalar "reward." We describe this framework, and define optimal behavior in this context.

A task is defined by a set of states, $s \in \mathcal{S}$, a set of actions, $a \in \mathcal{A}$, a state-action transition function (mapping state/action pairs to the resulting state), $T : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$, and a reward function (mapping state/action pairs to a scalar reward), $R : \mathcal{S} \times \mathcal{A} \rightarrow \Re$. At each time step, the learner (also called the *agent*) selects an action, and then as a result is given a reward and transitions to a new state. The goal of reinforcement learning is to learn a *policy*, a mapping from states to actions, $\pi : \mathcal{S} \rightarrow \mathcal{A}$, that maximizes the sum of its reward over time. The most common formulation of "reward over time" is a discounted sum of rewards into an infinite future. We use the infinite-horizon discounted model where reward over time is a geometrically discounted sum in which the discount , $0 \leq \gamma < 1$, devalues rewards received in the future. Accordingly, when following policy $\pi$, we can define the *value* of each state to be:

$$V^{\pi}(s) = \sum_{t=0}^{\infty} \gamma^t r_t, \tag{1}$$

where $r_t$ is the reward received $t$ time steps after starting in state $s$. The optimal policy, written $\pi^\star$, is the one that maximizes the value, $V^{\pi}(s)$, over all states $s$.

In order to learn the optimal policy, we learn its value function, $V^\star$, and its more specific correlate, called $Q$. Let $Q^\star(s, a)$ be the value of selecting action $a$ from state $s$, and thereafter following the optimal policy. This is expressed as:

$$Q^{\star}(s,a) = R(s,a) + \gamma V^{\star}(T(s,a)). \quad (2)$$

We can now define the optimal policy in terms of $Q^{\star}$ by selecting from each state the action with the highest expected future reward: $\pi^{\star}(s) = \arg\max_a Q^{\star}(s,a)$. The seminal work by Bellman (1957) shows that the optimal policy can be found straightforwardly by dynamic programming.

### 3.2. Spidering as Reinforcement Learning

As an aid to understanding how reinforcement learning relates to spidering, consider the common reinforcement learning task of a mouse exploring a maze to find several pieces of cheese. The mouse can perform actions for moving among the grid squares of the maze. The mouse receives a reward for finding each piece of cheese. The state is both the position of the mouse and the locations of the cheese pieces remaining to be consumed (since the cheese can only be consumed and provide reward once). Note that the mouse only receives immediate reward for finding a maze square containing cheese, but that in order to act optimally it must choose actions based on future rewards as well.

In the spidering task, the on-topic documents are immediate rewards, like the pieces of cheese. The actions are following a particular hyperlink. The state is the set of on-topic documents that remain to be consumed, and the set of URLs that have been encountered.[2] The state does not include the current "position" of the agent since a crawler can go next to any URL it has previously encountered. The number of actions is large and dynamic, in that it depends on which pages the spider has visited so far.

The most important features of topic-specific spidering that make reinforcement learning an especially good framework for defining the optimal solution are: (1) performance is measured in terms of reward over time because it is better to locate on-topic documents sooner, given time limitations, and (2) the environment presents situations with delayed reward, in that on-topic documents may be several hyperlink traversals away from the current choice point.

[2] It is as if the mouse can jump to any square, as long as it has already visited a bordering square. Thus the state is not a single position, but the position and shape of the boundary.

### 3.3. Practical Approximations

The problem now is how to apply reinforcement learning to spidering in such a way that it can be practically solved. Unfortunately, the state space is huge: exponential in the number of on-topic documents on the Web. The action space is also large: the number of unique hyperlinks that the spider could possibly visit.

In order to make learning feasible we use value function approximation. That is, we train a learning algorithm that generalizes across states and is able to predict the $Q$-value of a previously unseen state/action pair. The spider that emerges from this training procedure efficiently explores new web graphs by estimating the expected future reward associated with new hyperlinks using this function approximator. The state space is so unusually large, however, that function approximation cannot support dynamic programming. Thus, like in work by Kearns, Mansour, and Ng (2000), we sample from the state space, and calculate a sum of expected future reward with an explicit *roll-out* solution using a model. The use of roll outs for policy evaluation is also used in TD-1 (Sutton, 1988).

We gather training data and build a model consisting of all the pages and hyperlinks found by exhaustively spidering a few web sites.[3] By knowing the complete web graph of the training data, we can easily define a near-optimal policy by automatic inspection of the web graph. We then execute that policy for a finite number of steps from state/action pairs for some subset of the states; these executions result in a sequence of immediate rewards. We then assign to these state/action pairs the $Q$-value calculated as the discounted sum of the reward sequence. These triplets of state, action and $Q$-value become the training data for our value function approximation.

In the next two sub-sections we describe the near-optimal policy on known web graphs, and the value function approximation.

### 3.4. Near-Optimal Policy on Known Hyperlink Graphs

Given full knowledge of a hyperlink graph built by exhaustively spidering a web site, it is straightforward to specify a near-optimal policy. The policy must choose to follow one hyperlink from among all the unfollowed hyperlinks that it knows about so far, the "fringe." At each time step, our near-optimal policy selects from the fringe the action that follows the hyperlink on the path to the closest immediate reward. For example, in Figure 4, the policy would choose action A at time 0

[3] This is the off-line version of our algorithm; the on-line version would be a form of policy improvement using roll-outs, as in Tesauro and Galperin (1997).

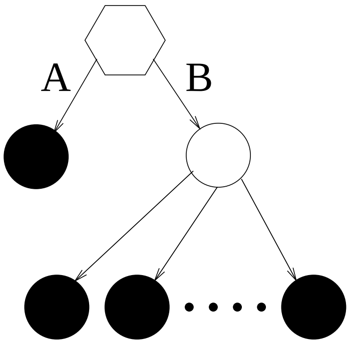



*Figure 4.* A representation of spidering space where arrows are hyperlinks and nodes are web documents. The hexagonal node represents an already-explored node; the circular nodes are unexplored. Filled-in circles denote the presence of immediate reward (target pages). When a spider is given the choice between an action that provides immediate reward and one that provides future reward, the spider always achieves the maximum discounted reward by choosing the immediate reward first. By first following $A$, the spider achieves rewards in the sequence 10111.... Following $B$ first only delays the first reward: 01111....

because it provides a reward at time 1, where choosing action B would delay the first immediate reward until time 2.

This policy closely approximates the optimal policy in cases where all non-zero immediate rewards have the same value. Figure 4 gives an example of a common spidering situation where our near-optimal policy makes the optimal decision. Here, the spider is given the option of taking actions $A$ and $B$. Since $A$ yields reward sooner, the near-optimal policy chooses this action. This near-optimal policy often makes the right decision. In fact, in the case that $\gamma \leq 0.5$, the only case where the policy may make a mistake is when two or more actions provide the first immediate reward equidistantly from the fringe. The heuristic policy arbitrarily selects one of these; in contrast, the optimal policy would select the hyperlink leading to the most additional reward, beyond just the first one.

We choose to begin with a near-optimal policy because simply *specifying* the optimal policy on a Web graph is a non-trivial optimization problem. We also believe that directly approximating the optimal policy would provide little practical benefit, since our near-optimal policy captures the optimal policy in many of the situations that a spider encounters.

### 3.5. Value Function Approximation

Using the above policy, the training procedure generates state/action/$Q$-value triples. As in most reinforcement learning solutions to problems with large state spaces, these triples then act as training data for supervised training of an approximation to the value function, $V(s)$, or a $Q$ function. To make this approximation we must specify which subset of states we use for training, the feature representation of a state and action, and the underlying learning algorithm to map features to $Q$-values. We choose a simple but intuitive set of states to use as training, map a state and hyperlink action to a set of words occurring around the hyperlink, and use naive Bayes to map words into a predicted $Q$-value.

For the experiments in this paper, we calculate the value of our near-optimal policy for all states where the fringe contains exactly one hyperlink. Thus, for each known hyperlink $a$, we estimate $Q(\{a\}, a)$ by roll-out to generate training data. Considering a larger set of state/action pairs might make our spidering framework impractical—taking advantage of a larger set would necessitate recalculating $Q$ values for every hyperlink that the spider follows.

The features of a state/action pair are a set of words. Given a hyperlink action $a$, the features are the neighboring words of $a$ on all previously visited pages in state $s$ where hyperlink $a$ occurs.[4] The precise definition of neighboring text is given for each data set is Section 3.6, but approximately it means words occurring near to the hyperlink on the page where it occurs. In many cases a unique hyperlink occurs on only one page. However, it is not uncommon that multiple pages contain the same hyperlink; in these cases we use the words on each of these multiple pages as our features.

Our value function approximator takes as inputs these words and gives an estimate of the $Q$-value. We perform this mapping by casting this regression problem as classification (Torgo & Gama, 1997). We discretize the discounted sum of future reward values of our training data into bins and treat each bin as a class. For each state/action pair we calculate the probabilistic class membership of each bin using naive Bayes (which is described in section 5.2.1). Then the $Q$-value of a new, unseen hyperlink is estimated by taking a weighted average of each bins' $Q$-value, using the probabilistic class memberships as weights.

All of the approximations that we have made are focused on ensuring that our framework is practical. The training phase has computational complexity $O(N)$, whereas the spidering phase is $O(N \log N)$ ($N$ is the number of hyperlinks). The $\log N$ term accounts for the need to sort the $Q$ values of those hyperlinks on the fringe. This term could be eliminated through an approximation such as discretizing the $Q$-value space. Hence, our framework does not significantly add to the computational

[4] Note that we are ignoring the part of the state that specifies which on-topic documents have already been consumed.

complexity of spidering. An efficient implementation should find Web page downloads to be the main bottleneck.

## 3.6. Experimental Results

In this section we provide empirical evidence that using reinforcement learning to guide the search of a spider increases its efficiency. We use two datasets, the Research Paper dataset, which is used in the Cora portal, and also the Corporate Officers dataset, where the goal is to locate specific company information.

### 3.6.1. *Datasets and Protocol*

In August 1998 we completely mapped the documents and hyperlinks of the web sites of computer science departments at Brown University, Cornell University, University of Pittsburgh and University of Texas. They include 53,012 documents and 592,216 hyperlinks. These web pages make up the Research Paper dataset. The target pages (for which a reward of 1 is given) are the 2,263 computer science research papers. They are identified with 95% precision by a simple hand-coded algorithm that locates abstracts and reference sections in postscript files with regular expressions. We perform a series of four test/train splits, in which the data from three universities is used to train a spider that is then tested on the fourth. The training data is used for value function approximation, as described in Section 3.5. In this dataset, the neighboring text for a URL is defined as the full text of the page where the URL is found with the anchor and nearby text marked specially. Each spidering run begins at the homepage of the test department. We report average performance across the four test sets.

In December 1998, we collected the Corporate Officers dataset, consisting of the complete web sites of 26 companies, totaling 6,643 web pages. The targets in this dataset are the web pages that include information about officers and directors of the company. One such page was located by hand for each company, giving a total of 26 target pages. We perform 26 test/train splits where each company's web site forms a test set, while the others are used for training. In this dataset, value function approximation proceeds by defining the neighboring text to be header and title words, the anchor text, portions of the URL itself (e.g. directory and file names) and a small set of words immediately before and after the hyperlink. Each spidering run begins at the homepage of the corresponding test company.

We present results of two different reinforcement learning spiders and compare them to a breadth-first-search spider. The first, Focused uses $\gamma = 0$, and closely mimics what is known as a "focused crawler."

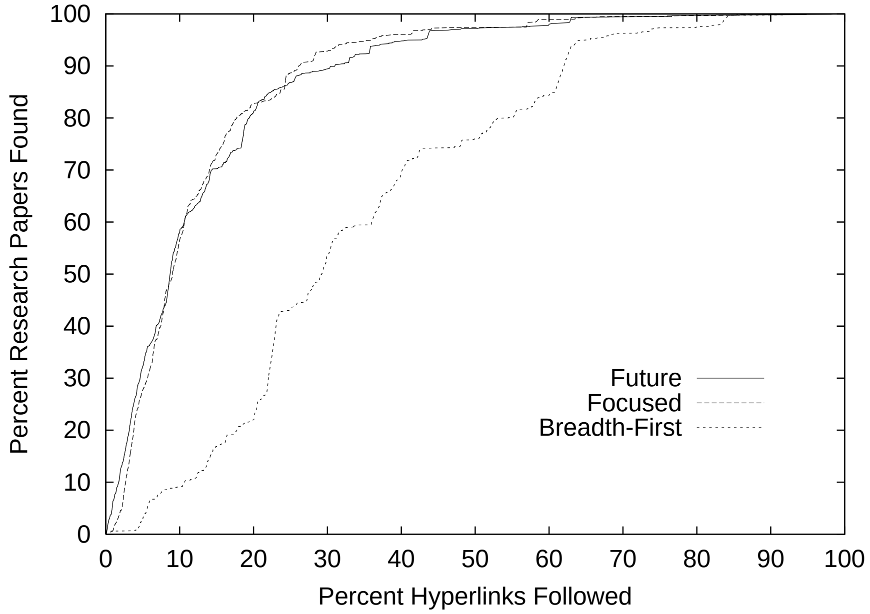



*Figure 5.* The performance of different spidering strategies, averaged over four test/train splits. The reinforcement learning spiders find target documents significantly faster than traditional breadth-first search.

(Chakrabarti et al., 1999) This spider employs a binary classifier that distinguishes between immediately relevant text and other text. Future uses $\gamma = 0.5$ and makes use of future reward, representing the $Q$-function with a more finely-discriminating multi-bin classifier. Here, training data is partitioned into bins based on the $Q$-value of each hyperlink. We found that a 3-bin classifier performed best on the Research Paper data while a 4-bin classifier yielded the best results on the Corporate Officers data.

3.6.2. *Finding Research Papers*

Results for the Research Paper dataset are depicted in Figures 5 and 6, comparing the three-bin Future spider against the two baselines. The number of research papers found is plotted against the number of pages visited, averaged over all four universities.

At all times during their search, both the Future and Focused spiders find significantly more research papers than breadth-first search. One measure of performance is the number of hyperlinks followed before 75% of the research papers are found. Both reinforcement learners are significantly more efficient, requiring exploration of less than 16% of the

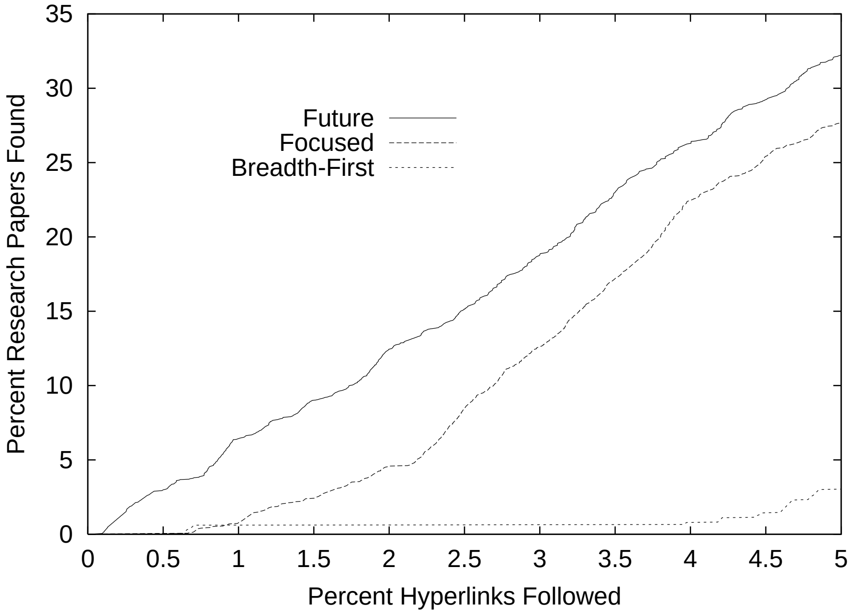



*Figure 6.* The performance of different spidering strategies during the initial stages of each spidering run. Here, the Future spider performs best, because identifying future rewards are crucial.

hyperlinks; in comparison, Breadth-first requires 48%. This represents a factor of three increase in spidering efficiency.

However, Future does not always perform as well as or better than Focused. In Figure 5, after the first 50% of the papers are found the Focused spider performs slightly better than Future. This is because the system has uncovered many links that will give immediate reward if followed, and the Focused spider recognizes them more accurately. In future work we are investigating techniques for improving classification to recognize these immediate rewards when the spider uses the larger number of bins required for regression with future reward.

We hypothesize that modeling future reward is more important when immediate reward is more sparse. While there is not significant separation between Focused and Future through most of the run, the early stages of the run provide a special environment; reward is very sparse, as most research papers lie several hyperlinks away from areas the spider has explored; subsequently, few immediate reward actions are available. Figure 6 shows the average performance of the spiders during the initial stages of spidering. We indeed see that Future, a spider which takes advantage of future rewards knowledge, does better than Focused. On average the Focused spider takes nearly three times as

Table I. A comparison of spidering performance on the Corporate Officers dataset. Each result shows the average percentage of each company's web site traversed before finding the goal page. Here, the 4 bin Future spider performs twice as well as Focused, and nearly three times as well as Breadth-First.

| Spidering Method | % Links Followed |
|---|---|
| Optimal | 3% |
| Future (4 bins) | 13% |
| Future (3 bins) | 22% |
| Future (5 bins) | 27% |
| Focused | 27% |
| Breadth-First | 38% |

long as Future to find the first 28 (5%) of the papers. While this result may seem insignificant at first, its importance becomes more clear in the Corporate Officer experiments described in the next section.

Through our Research Papers experiments, we have shown that our reinforcement learning framework has promise: it significantly outperforms breadth-first search, performs much like a focused crawler overall and outperforms a focused spider in the important early stages. The Corporate Officers dataset is more extreme in its reward sparsity, and shows this improved performance more dramatically.

#### 3.6.3. *Finding Corporate Officers*

Table I shows spidering results on the Corporate Officers dataset. The calculated figure is the average percent of each company's web site the spider traversed before finding the single goal. On average, the four-bin Future spider is able to locate the goal page after traversing only 13% of the hyperlinks. This is twice as efficient as Focused, which follows an average of 27% of the hyperlinks before locating the target page. In further contrast, Future performs three-times as efficient as the Breadth-First spider, which follows an average of 38% of the hyperlinks before finding the goal page.

Each spidering run entails locating a single Web page within a corporate web site. In our experiments, the sites ranged from 20 to almost 1000 web pages. In contrast to the Research Paper dataset, where the number of Web pages per goal page is 23, the Corporate Officers dataset contains 256 web pages per goal page, a significant increase in sparsity. As a result, two instantiations of the Future spider perform significantly better than the Focused spider. Since Future and Focused are otherwise identical, this added efficiency must come from Future's knowledge of future reward.

While the three- and four-bin Future spiders outperform Focused, there is a tradeoff between the flexibility of the classifier-regressor and classification accuracy. Experiments with a five-bin classifier result in worse performance—roughly equivalent to the Focused spider, following an average of 27% of available hyperlinks before locating the target page. While additional bins can provide a stronger basis for $Q$-value prediction, they also create a more complicated classification task; more bins generally decrease classification accuracy. Hence, we reason that our naive Bayes classifier cannot take advantage of the additional bin in the 5 bin Future spider. Better features and other methods for improving classifier accuracy (such as shrinkage (McCallum, Rosenfeld, Mitchell, & Ng, 1998)) should allow the more sensitive multi-bin classifier to perform better.

These results indicate that when there are many more non-target pages than target pages, (*i.e.* reward is sparse), the Future spider's explicit modeling of future reward significantly increases its efficiency over the Focused spider. By tuning the tradeoffs appropriately, we should be able to achieve increased performance, even when reward is less sparse.

The construction of a topic-specific portal, such as Cora, requires the location of large quantities of relevant documents. However, such documents are often sparsely distributed throughout the Web. As the Internet continues to grow and domain-specific search services become more popular, it will become increasingly important that spiders be able to gather on-topic documents efficiently. The spidering work presented here is an initial step towards creating such efficient spidering. We believe that further understanding of the reinforcement learning framework and the relaxation of the simplifying assumptions used here will lead to additional improvements in the future.

## 4. Information Extraction

Information extraction is concerned with identifying phrases of interest in textual data. For many applications, extracting items such as names, places, events, dates, and prices is a powerful way to summarize the information relevant to a user's needs. In the case of a domain-specific portal, the automatic identification of important information can increase the accuracy and efficiency of a directed query.

In Cora we use hidden Markov models (HMMs) to extract the fields relevant to computer science research papers, such as titles, authors, affiliations and dates. One HMM extracts information from each paper's header (the words preceding the main body of the paper). A second HMM processes the individual references in each paper's reference section. The extracted text segments are used (1) to allow searches over specific fields, (2) to provide useful, effective presentation of search results (*e.g.* showing title in bold), and (3) to match references to papers during citation grouping.

Our research interest in HMMs for information extraction is particularly focused on learning the appropriate state and transition structure of the models from training data, and estimating model parameters with labeled and unlabeled data. We show that models with structures learned from data outperform models built with one state per extraction class. We also demonstrate that using distantly-labeled data for parameter estimation improves extraction accuracy, but that Baum-Welch estimation of model parameters with unlabeled data degrades performance.

### 4.1. Hidden Markov Models

Hidden Markov modeling is a powerful statistical machine learning technique that is just beginning to gain use in information extraction tasks (*e.g.* Leek, 1997; Bikel et al., 1997; Freitag & McCallum, 1999). HMMs offer the advantages of having strong statistical foundations that are well-suited to natural language domains and robust handling of new data. They are also computationally efficient to develop and evaluate due to the existence of established training algorithms. The disadvantages of using HMMs are the need for an *a priori* notion of the model topology and, as with any statistical technique, a sufficient amount of training data to reliably estimate model parameters.

Discrete output, first-order HMMs are composed of a set of states $Q$, with specified initial and final states $q_I$ and $q_F$, a set of transitions between states $(q \rightarrow q')$, and a discrete vocabulary of output symbols $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_M\}$. The model generates a string $\mathbf{w} = w_1 w_2 \ldots w_l$ by beginning in the initial state, transitioning to a new state, emitting an output symbol, transitioning to another state, emitting another symbol, and so on, until a transition is made into the final state. The parameters of the model are the transition probabilities $\mathrm{P}(q \rightarrow q')$ that one state follows another and the emission probabilities $\mathrm{P}(q \uparrow \sigma)$ that a state emits a particular output symbol. The probability of a string $\mathbf{w}$ being emitted by an HMM $M$ is computed as a sum over all possible paths by:

$$\mathrm{P}(\mathbf{w}|M) = \sum_{q_1,\ldots,q_l \in Q^l} \prod_{k=1}^{l+1} \mathrm{P}(q_{k-1} \rightarrow q_k)\mathrm{P}(q_k \uparrow w_k), \tag{3}$$

where $q_0$ and $q_{l+1}$ are restricted to be $q_I$ and $q_F$ respectively, and $w_{l+1}$ is an end-of-string token. The Forward algorithm can be used to calculate this probability efficiently (Rabiner, 1989).

The observable output of the system is the sequence of symbols that the states emit, but the underlying state sequence itself is hidden. One common goal of learning problems that use HMMs is to recover the state sequence $V(\mathbf{w}|M)$ that has the highest probability of having produced an observation sequence:

$$V(\mathbf{w}|M) = \arg\max_{q_1 \ldots q_l \in Q^l} \prod_{k=1}^{l+1} \mathrm{P}(q_{k-1} \rightarrow q_k)\mathrm{P}(q_k \uparrow w_k). \tag{4}$$

Fortunately, the Viterbi algorithm (Viterbi, 1967) efficiently recovers this state sequence.

### 4.2. HMMs for Information Extraction

Hidden Markov models provide a natural framework for modeling the production of the headers and references of research papers. They explicitly represent extraction classes as states, efficiently model the frequencies of word occurrences for each class, and take class sequence into account. We want to label each word of a header or reference as belonging to a class such as title, author, journal, or keyword. We do this by modeling the entire header or reference (and all of the classes to extract) with one HMM. This task varies from the more classic extraction task of identifying a small set of target words from a large document containing mostly uninformative text.

HMMs may be used for information extraction by formulating a model in the following way: each state is associated with a class that we want to extract, such as title, author or affiliation. Each state emits words from a class-specific multinomial (unigram) distribution. We can learn the class-specific multinomial distributions and the state transition probabilities from training data. In order to label a new header or reference with classes, we treat the words from the header or reference as observations and recover the most-likely state sequence with the Viterbi algorithm. The state that produces each word is the class tag for that word. An example HMM for headers, annotated with class labels and transition probabilities, is shown in Figure 7.

Hidden Markov models, while relatively new to information extraction, have enjoyed success in related natural language tasks. They have been widely used for part-of-speech tagging (Kupiec, 1992), and have more recently been applied to topic detection and tracking (Yamron, Carp, Gillick, Lowe, & van Mulbregt, 1998) and dialog act modeling (Stolcke, Shriberg, Bates, Coccaro, Jurafsky, Martin, Meteer, Ries, Tay-

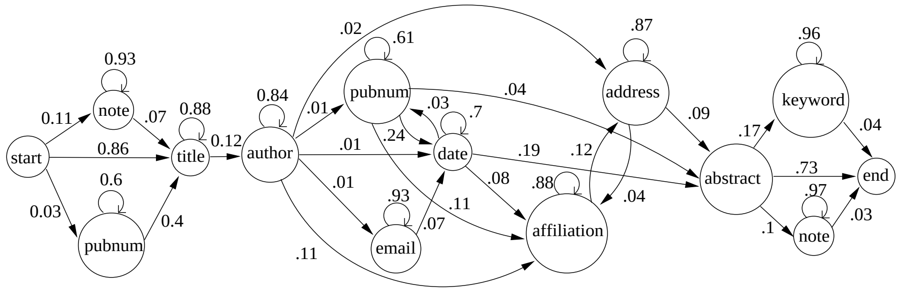



*Figure 7.* Example HMM for the header of a research paper. Each state emits words from a class-specific multinomial distribution.

lor, & Ess-Dykema, 1998). Other systems using HMMs for information extraction include those by Leek (1997), who extracts gene names and locations from scientific abstracts, and the Nymble system (Bikel et al., 1997) for named-entity extraction. Unlike our work, these systems do not consider automatically determining model structure from data; they either use one state per class, or use hand-built models assembled by inspecting training examples. Freitag and McCallum (1999) hand-build multiple HMMs, one for each field to be extracted, and focus on modeling the immediate prefix, suffix, and internal structure of each field. In contrast, we focus on learning the structure of one HMM to extract all the relevant fields, which incorporates the observed sequences of extraction fields directly in the model.

#### 4.2.1. *Learning model structure from data*

In order to build an HMM for information extraction, we must first decide how many states the model should contain, and what transitions between states should be allowed. A reasonable initial model is to use one state per class, and to allow transitions from any state to any other state (a fully-connected model). However, this model may not be optimal in all cases. When a specific hidden sequence structure is expected in the extraction domain, we may do better by building a model with multiple states per class, with only a few transitions out of each state. Such a model can make finer distinctions about the likelihood of encountering a class at a particular location in the document, and can model specific local emission distribution differences between states of the same class. For example, in Figure 7, there are two states for the "publication number" class, which allows the class to exhibit different transition behavior depending on where in the header the class is encountered; if a publication number is seen before the title, we would expect transitions from and to a different set of states than if it is seen after the author names. Likewise, the HMM has two states for

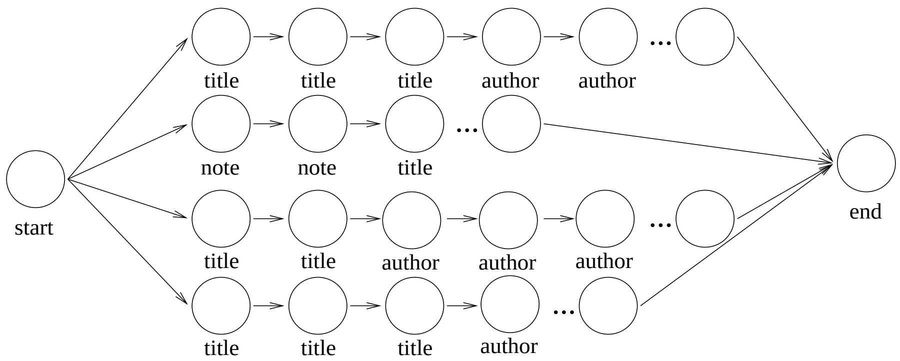



*Figure 8.* Example of a maximally specific HMM built from four training instances, which is used as the starting point for state merging.

the "note" class. These two states, although from the same class, may benefit from different emission distributions, due to the different types of copyright and publication notes that occur at the beginning and end of a header.

An alternative to simply assigning one state per class is to learn the model structure from training data. Training data labeled with class information can be used to build a maximally-specific model. An example of this model built from just four labeled examples is shown in Figure 8. Each word in the training data is assigned its own state, which transitions to the state of the word that follows it. Each state is associated with the class label of its word token. A transition is placed from the start state to the first state of each training instance, as well as between the last state of each training instance and the end state.

This model can be used as the starting point for a variety of state merging techniques. We propose two simple types of merges that can be used to generalize the maximally-specific model. First, "neighbor-merging" combines all states that share a transition and have the same class label. As multiple neighbor states with the same class label are merged into one, a self-transition loop is introduced, whose probability represents the expected state duration for that class. For example, in Figure 8, the three adjacent title states from the first header would be merged into a single title state, which would have a self-transition probability of 2/3.

Second, "V-merging" merges any two states that have the same label and share transitions from or to a common state. V-merging reduces the branching factor of the maximally-specific model. We apply V-merging to models that have already undergone neighbor-merging. For example, again in Figure 8, instead of selecting from among three transitions from the start state into title states, the V-merged model would merge the children title states into one, so that only one transition from the start state to the title state would remain. The V-merged model can be used for extraction directly, or more state merges can be made automatically or by hand to generalize the model further.

4.2.2. *Labeled, unlabeled, and distantly-labeled data*

Once a model structure has been selected, the transition and emission parameters need to be estimated from training data. While obtaining unlabeled training data is generally not too difficult, acquiring labeled training data is more problematic. Labeled data is expensive and tedious to produce, since manual effort is involved. It is also valuable, since the counts of class transitions $N(q \to q')$ and the counts of a word occurring in a class $N(q \uparrow \sigma)$ can be used to derive maximum likelihood estimates for the parameters of the HMM:

$$\hat{\mathrm{P}}(q \to q') = \frac{N(q \to q')}{\sum_{s \in Q} N(q \to s)}, \tag{5}$$

$$\hat{\mathrm{P}}(q \uparrow \sigma) = \frac{N(q \uparrow \sigma)}{\sum_{\rho \in \Sigma} N(q \uparrow \rho)}. \tag{6}$$

Smoothing of the distributions is often necessary to avoid probabilities of zero for the transitions or emissions that do not occur in the training data. Absolute discounting and additive smoothing are examples of possible smoothing strategies. Chen and Goodman (1998) provide a thorough discussion and comparison of different smoothing techniques.

Unlabeled data, on the other hand, can be used with the Baum-Welch training algorithm (Baum, 1972) to train model parameters. The Baum-Welch algorithm is an iterative Expectation-Maximization (EM) algorithm that, given an initial parameter configuration, adjusts model parameters to locally maximize the likelihood of unlabeled data. Baum-Welch training suffers from the fact that it finds local maxima, and is thus sensitive to initial parameter settings.

A third source of valuable training data is what we refer to as *distantly-labeled* data. Sometimes it is possible to find data that is labeled for another purpose, but which can be partially applied to the domain at hand. In these cases, it may be that only a portion of the labels are relevant, but the corresponding data can still be added into the model estimation process in a helpful way. For example, BibTeX files are bibliography databases that contain labeled citation information. Several of the labels that occur in citations, such as title and author, also occur in the headers of papers, and this labeled data can be used in training emission distributions for header extraction. However, other BibTeX fields are not relevant to the header extraction task, and not all of the header fields occur in the BibTeX data. In addition, the data does not include any information about sequences of classes in headers and therefore cannot be used for transition distribution estimation.

Class emission distributions can be trained directly using either the labeled training data (L), a combination of the labeled and distantly-labeled data (L+D), or a linear interpolation of the labeled and distantly-labeled data (L*D). In the L+D case, the word counts of the labeled and distantly-labeled data are pooled together before deriving the emission distributions. In the L*D case, separate emission distributions are trained for the labeled and distantly-labeled data, and then the two distributions are interpolated together using a mixture weight derived from Expectation-Maximization of the labeled data, where each word of the labeled data is left out of the maximum likelihood calculation in turn. These three cases are shown below:

$$\hat{\mathrm{P}}_L(w_i) = \frac{f(N_L(w_i))}{\sum_{i=1}^{V} N_L(w_i)} \tag{7}$$

$$\hat{\mathrm{P}}_{L+D}(w_i) = \frac{f(N_L(w_i) + N_D(w_i))}{\sum_{i=1}^{V} N_L(w_i) + N_D(w_i)} \tag{8}$$

$$\hat{\mathrm{P}}_{L*D}(w_i) = \lambda \hat{\mathrm{P}}_L(w_i) + (1-\lambda)\hat{\mathrm{P}}_D(w_i), \tag{9}$$

where $N(w_i)$ is the count of word $w_i$ in the class, $\lambda$ is the mixture weight, and $f()$ represents a smoothing function, used to avoid probabilities of zero for the vocabulary words that are not observed for a particular class.

### 4.3. Experimental Results

We focus our information extraction experiments on extracting relevant information from the headers of computer science research papers, though the techniques described here apply equally well to reference extraction. We define the header of a research paper to be all of the words from the beginning of the paper up to either the first section of the paper, usually the introduction, or to the end of the first page, whichever occurs first. The abstract is automatically located using regular expression matching and changed to a single 'abstract' token. Likewise, an 'intro' or 'page' token is added to the end of each header to indicate whether a section or page break terminated the header. A few special classes of words are identified using simple regular expressions and converted to special identifying tokens: email addresses, web addresses, year numbers, zip codes, technical report numbers, and all other numbers. All punctuation, case and newline information is removed from the text.

Table II. Mixture weights for the labeled (L) and distantly-labeled (D) data for the L*D emission distributions. Mixture weights are derived from Expectation-Maximization of the labeled data, where each word of the labeled data is left out of the maximum likelihood calculation in turn.

| Class | L | D | Class | L | D | Class | L | D |
|---|---|---|---|---|---|---|---|---|
| Address | 0.861 | 0.139 | Date | 0.909 | 0.091 | Note | 0.808 | 0.192 |
| Affiliation | 0.858 | 0.142 | Email | 0.657 | 0.343 | Title | 0.329 | 0.671 |
| Author | 0.446 | 0.554 | Keyword | 0.470 | 0.530 | Web | 0.629 | 0.371 |

The target classes we wish to identify include the following fifteen categories: title, author, affiliation, address, note, email, date, abstract, introduction (intro), phone, keywords, URL, degree, publication number (pubnum), and page. The abstract, intro and page classes are each represented by a state that outputs only the token of that class. The degree class captures the language associated with Ph.D. or Master's theses, such as "submitted in partial fulfillment of..." and "a thesis by...". The note field commonly accounts for phrases from acknowledgements, copyright notices, and citations.

A set of research papers were selected at random from the Cora repository. The header of each paper was identified, and a 500-header, 23,557 word token training set and a 407-header, 18,863 word token test set were formed. Distantly-labeled training data was acquired from 176 BibTeX files that were collected from the Web. These files consist of 2.4 million words, which contribute to the following nine header classes: address, affiliation, author, date, email, keywords, note, title, and URL.

For each emission distribution training case (L, L+D, L*D), a fixed vocabulary is derived from all of the words in the training data used. The labeled data results in a 4,914-word vocabulary, and the labeled and distantly-labeled data together contain 92,426 distinct words. Absolute discounting (Ney, Essen, & Kneser, 1994) is used as the smoothing function. An unknown word token is added to the vocabularies to model out-of-vocabulary words. Any words in the testing data that are not in the vocabulary are mapped to this token. The probability of the unknown word is estimated separately for each class, and is assigned a portion of the discount mass proportional to the fraction of singleton words observed only in the current class. In the L*D case, the mixture weight is derived from Expectation-Maximization of the labeled data, and indicates the relative importance of each information source to the predictive abilities of the combined distribution. Mixture weights for the nine classes with distantly-labeled data are given in Table II. Most

Table III. Word extraction error rate (%) for models with one state per class when the emission parameters are estimated from labeled (L) data, a combination of labeled and distantly-labeled (L+D) and an interpolation of labeled and distantly-labeled (L*D) data. Using the interpolation of the two data sources provides the best extraction performance.

| Model | # States | # Transitions | L | L+D | L*D |
|---|---|---|---|---|---|
| Full | 17 | 255 | 37.3 | 42.7 | 35.6 |
| Self | 17 | 252 | 14.4 | 17.1 | 10.7 |
| ML | 17 | 149 | 9.9 | 10.8 | 7.9 |
| Smooth | 17 | 255 | 10.4 | 11.5 | 8.2 |

classes give a higher weight to the labeled data, with the exception of the author, keyword and title classes, which assign more value to the distantly-labeled data.

We build several HMM models, varying model structures and training conditions, and test the models by finding the Viterbi paths for the test set headers. Performance is measured by word classification error, which is the percentage of header words emitted by a state with a different label than the words' true label.

#### 4.3.1. *Model selection – One state per class*

The first set of models each use one state per class. Emission distributions are trained for each class on either the labeled data (L), the combination of the labeled and distantly-labeled data (L+D), or the interpolation of the labeled and distantly-labeled data (L*D). Extraction results for these models are reported in Table III.

The Full model is a fully-connected model where all transitions are assigned uniform probabilities. It relies only on the emission distributions to choose the best path through the model, and results in an error rate of 35.6%. The Self model is similar, except that the self-transition probability is set according to the maximum likelihood estimate from the labeled data, with all other transitions set uniformly. This model benefits from the additional information of the expected number of words to be emitted by each state, and its error rate drops to 10.7%. The ML model sets all transition parameters to their maximum likelihood estimates, and achieves the lowest error of 7.9% among this set of models. The Smooth model adds an additional smoothing count of one to each transition, so that all transitions have non-zero probabilities, but smoothing the transition probabilities does not lower the error rate. For all models, the combination of the labeled and distantly-labeled

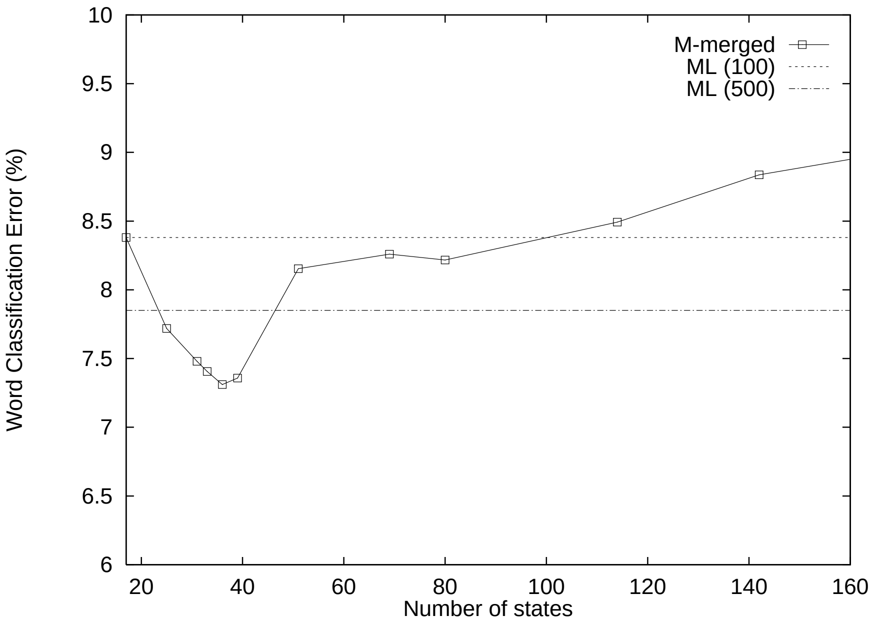



*Figure 9.* Extraction error for multi-state models as states are merged. The dashed lines represent the baseline performances of the ML model, trained on all 500 headers and on the same 100 headers as the multi-state models.

data (L+D) negatively affects performance relative to the labeled data results. However, the interpolation of the distantly-labeled data with the labeled data (L*D) consistently provides several percentage points decrease in error over training on the labeled data alone. We will refer back to the ML model results in the next comparisons, as the best representative of the models with one state per class.

4.3.2. *Model selection – Deriving structure from data*

The next set of models are learned from data; both the number of states and the transitions between the states are derived by state merging techniques. We first consider models built from a combination of automated and manual techniques. Starting from a neighbor-merged model of 805 states built from 100 randomly selected labeled training headers, states with the same class label are manually merged in an iterative manner. The manual merges are performed by a domain expert, and only 100 of the 500 headers are used in order to keep the manual state selection process manageable. Transition counts are preserved throughout the merges to estimate maximum likelihood transition probabilities. Each state uses its smoothed class emission distribution estimated from the interpolation of the labeled and distantly-labeled data (L*D). Extraction performance, measured as the number of states decreases

Table IV. Word extraction error rate (%) for models learned from data compared to the best model that uses one state per class. Emission probabilities are estimated in three different ways.

| Model | # States | # Transitions | L | L+D | L*D |
|---|---|---|---|---|---|
| ML | 17 | 149 | 9.9 | 10.8 | 7.9 |
| M-merged | 36 | 164 | 9.0 | 9.7 | 7.3 |
| V-merged | 155 | 402 | 9.7 | 10.5 | 7.6 |

during the merging, is plotted in Figure 9. The dashed lines on the figure represent the baseline performances of the ML model, trained on all 500 headers and on the same 100 headers as the multi-state models. The models with multiple states per class outperform the ML model for both training conditions, particularly when 30 to 40 states are present. In fact, when trained on the same amount of data, the multi-state models outperform the ML model for any number of states greater than one per class and less than 100. The best performance of 7.3% is obtained by the model containing 36 states. We refer to this model as the M-merged model. This result shows that more complex model structure benefits extraction performance of HMMs on the header task.

We compare this result to the performance of a 155-state V-merged model created entirely automatically from all of the labeled training data. A summary of the results of the ML model, the M-merged model, and the V-merged model is presented in Table IV. Once again, the L*D results are superior to the L and L+D results. In addition, both the M-merged and V-merged models outperform the ML model by a statistically significant margin in the L*D case, as determined with McNemar's test ($p < 0.005$ each).

Table V provides a closer look at the errors in each class for the ML, M-merged and V-merged models when using emission distributions trained on labeled (L) and interpolated (L*D) data. Classes for which there is distantly-labeled training data are indicated in bold. For several of the classes, such as title and author, there is a noticeable decrease in error when the distantly-labeled data is included. The poorest performing individual classes are the degree, publication number, and URL classes. The URL class has a particularly high error for the M-merged model, when limited URL class examples in the 100 training headers probably kept the URL state from having transitions to and from as many states as necessary.

Table V. Individual word extraction error rates (%) for each class with the ML, M-merged and V-merged models. Classes noted in bold occur in distantly-labeled data. The Abstract class can have a non-zero error due to emission distribution smoothing for the other classes.

| Class | # words | ML | | M-merged | | V-merged | |
|---|---|---|---|---|---|---|---|
| | | L | L*D | L | L*D | L | L*D |
| Abstract | 349 | 0 | 0 | 1.7 | 1.4 | 0.3 | 0.3 |
| **Address** | 2058 | 4.3 | 4.6 | 4.9 | 5.0 | 4.3 | 4.5 |
| **Affiliation** | 3429 | 12.7 | 8.9 | 12.1 | 9.4 | 12.2 | 8.9 |
| **Author** | 2543 | 4.7 | 2.4 | 5.2 | 2.9 | 4.8 | 2.4 |
| **Date** | 265 | 2.6 | 3.4 | 3.4 | 3.0 | 2.6 | 3.8 |
| Degree | 462 | 24.2 | 29.2 | 19.7 | 26.8 | 22.9 | 27.5 |
| **Email** | 473 | 11.0 | 11.2 | 12.9 | 13.5 | 11.0 | 11.0 |
| **Keyword** | 895 | 8.4 | 2.0 | 2.9 | 1.2 | 5.9 | 1.1 |
| **Note** | 4489 | 15.7 | 15.4 | 12.3 | 11.3 | 15.5 | 14.3 |
| Phone | 160 | 6.3 | 6.9 | 10.6 | 13.1 | 6.9 | 8.1 |
| Pubnum | 131 | 35.1 | 35.1 | 38.9 | 39.7 | 35.1 | 35.1 |
| **Title** | 3177 | 6.4 | 1.6 | 6.6 | 2.1 | 7.0 | 1.8 |
| **URL** | 36 | 19.4 | 16.7 | 58.3 | 58.3 | 36.1 | 33.3 |
| Overall | 18863 | 9.9 | 7.9 | 9.0 | 7.3 | 9.7 | 7.6 |

4.3.3. *Incorporating Unlabeled Data*

Next, we demonstrate that using unlabeled data for parameter estimation does not help classification accuracy for this extraction task. Baum-Welch training, the standard technique for estimating HMM parameters from unlabeled data, produces new transition and emission parameter values that locally maximize the likelihood of the unlabeled data. Careful selection of the initial parameter values is thus essential for finding a good local maximum.

Five thousand unlabeled headers, composed of 287,770 word tokens are used as training data. Baum-Welch training is run on the ML and M-merged models. Model parameters are initialized to the maximum likelihood transition probabilities from the labeled data and the interpolated (L*D) emission distributions. The models are tested under three different conditions; the extraction results, as well as the model perplexities on the test set, are shown in Table VI. Perplexity is a measure of how well the HMMs model the data; a lower value indicates a model that assigns a higher likelihood to the observations from the test set.

Table VI. Word extraction error rate (%) and test set perplexity (PP) for the ML and M-merged models after Baum-Welch training.

| | ML | | M-merged | |
|---|---|---|---|---|
| | Error | Perplexity | Error | Perplexity |
| Initial (L*D) | 7.9 | 475 | 7.3 | 486 |
| BW-uniform | 10.0 | 374 | 10.9 | 363 |
| BW-varied | 10.6 | 369 | 11.9 | 357 |

The "Initial" result is the performance of the models using the initial parameter estimates. These results are the same as the L*D case in Table IV. After Baum-Welch training, the vocabulary words that do not occur in the unlabeled data are given a probability of zero in the newly-estimated emission distributions. Thus, the new emission distributions need to be smoothed; we choose to do this by interpolating them with the initial parameter estimates. Each state's newly-estimated emission distribution is linearly interpolated with its initial distribution using a mixture weight of $\lambda$. The "BW-uniform" result shows performance when Baum-Welch training has been run using all of the unlabeled training data, and the mixture weights for the initial and newly-estimated emission distributions are set uniformly to 0.5 each. In the "BW-varied" case, mixture weights are optimized separately for each state. Ninety percent of the unlabeled data is used for Baum-Welch training. The newly-estimated emission distributions are then interpolated with the initial emission distributions using uniform mixture weights. One iteration of the Baum-Welch algorithm is run over the remaining 10% of the unlabeled data to assign expected words counts to each state. These expected word counts are used with the EM algorithm to set the mixture weights for each state individually.

Baum-Welch training degrades classification performance for both the ML and M-merged models. The lack of improvement in classification accuracy can be partly explained by the fact that Baum-Welch training maximizes the likelihood of the unlabeled data, not the classification accuracy. However, Baum-Welch training does result in improved predictive modeling of the header domain. This improvement is pointed out through the decrease in test set perplexity. The perplexity of the test set improves over the initial settings with Baum-Welch re-estimation, and improves even further with careful selection of the emission distribution mixture weights. Merialdo (1994) finds a similar effect on tagging accuracy when training part-of-speech taggers using Baum-Welch training when starting from well-estimated initial parameter estimates.

### 4.4. Discussion

Our experiments show that hidden Markov models do well at extracting important information from the headers of research papers. We achieve a low error rate of 7.3% over all classes of the headers, and class-specific error rates of 2.1% for titles and 2.9% for authors. We have demonstrated that models that contain multiple states per class do provide increased extraction accuracy over models that use only one state per class. This improvement is due to more specific transition context modeling that is possible with more states. We expect that it is also beneficial to have localized emission distributions, which can capture distribution variations that are dependent on the position of the class in the header.

Distantly-labeled data has proven to be valuable in providing robust parameter estimates. The interpolation of distantly-labeled data provides a consistent decrease in extraction error for headers. In cases where little labeled training data is available, distantly-labeled data is a helpful resource.

The high accuracy of our header extraction results allows Cora to process and present search results effectively. The success of these extraction techniques is not limited to this single application, however. For example, applying these techniques to reference extraction achieves a word extraction error rate of 6.6%. These techniques are also applicable beyond the domain of research papers. We have shown how distantly-labeled data can improve extraction accuracy; this data is available in electronic form for many other domains. For example, lists of names (with relative frequencies) are provided by the U.S. Census Bureau, street names and addresses can be found in online phone books, and discussion groups and news sites provide focused, topic-specific collections of text. These sources of data can be used to derive class-specific words and relative frequencies, which can then be applied to HMM development for a vast array of domain-specific portals.

## 5. Classification into a Topic Hierarchy

Topic hierarchies are an efficient way to organize, view and explore large quantities of information that would otherwise be cumbersome. The U.S. Patent database, Yahoo, MedLine and the Dewey Decimal system are all examples of topic hierarchies that exist to make information more manageable.

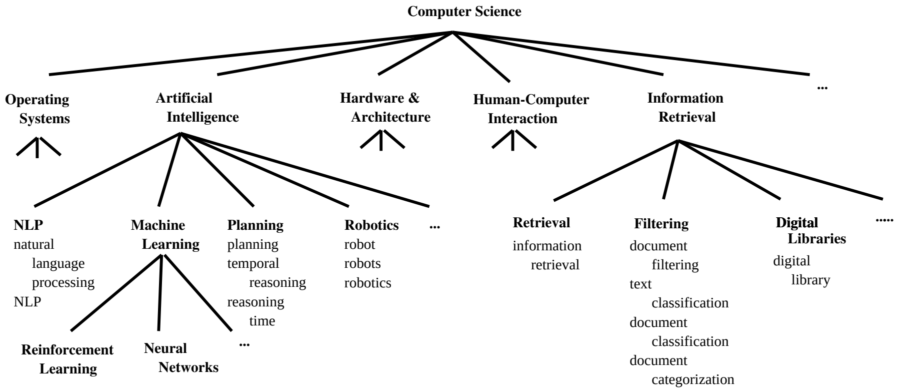



*Figure 10.* A subset of Cora's computer science hierarchy with the complete keyword list for each of several categories. These keywords are used to initialize bootstrapping.

As Yahoo has shown, a topic hierarchy can be a useful, integral part of a portal. Many search engines (*e.g.* AltaVista, Google and Lycos) now display hierarchies on their front page. This feature is equally or more valuable for domain-specific Internet portals. We have created a 70-leaf hierarchy of computer science topics for Cora, part of which is shown in Figures 1 and 10.

A difficult and time-consuming part of creating a hierarchy is populating it with documents by placing them into the correct topic nodes. Yahoo has hired large numbers of people to categorize web pages into their hierarchy. The U.S. patent office also employs people to perform the job of categorizing patents. In contrast, we automate the process of placing documents into leaf nodes of the hierarchy with learned text classifiers.

Traditional text classification algorithms learn representations from a set of labeled data. Unfortunately, these algorithms typically require on the order of hundreds of examples per class. Since labeled data is tedious and expensive to obtain, and our class hierarchy is large, using the traditional supervised approach is not feasible. In this section we describe how to create a text classifier by bootstrapping without any labeled documents, using only a few keywords per class and a class hierarchy. Both of these information sources are easily obtained. Keywords are quicker to generate than even a small number of labeled documents. Many classification problems naturally come with hierarchically-organized classes.

Bootstrapping is a general framework for iteratively improving a learner using unlabeled data. Bootstrapping is initialized with a small amount of seed information that can take many forms. Each iteration has two steps: (1) labels are estimated for unlabeled data from the currently learned model, and (2) the unlabeled data and these estimated labels are incorporated as training data into the learner. Bootstrapping approaches have been used for information extraction (Riloff & Jones, 1999), word sense disambiguation (Yarowsky, 1995), and hypertext classification (Blum & Mitchell, 1998).

Our algorithm for text classification is initialized by using keywords to generate preliminary labels for some documents by term-matching. The bootstrapping iterations are EM steps that use unlabeled data and hierarchical shrinkage to estimate parameters of a naive Bayes classifier. An outline of the entire algorithm is presented in Table VII. In experimental results, we show that the learned classifier has accuracy that approaches human agreement levels for this domain.

### 5.1. Initializing Bootstrapping with Keywords

The initialization step in the bootstrapping process uses keywords to generate preliminary labels for as many of the unlabeled documents as possible. For each class a few keywords are generated by a human trainer. Figure 10 shows examples of the number and type of keywords selected for our experimental domain.

We generate preliminary labels from the keywords by term-matching in a rule-list fashion: for each document, we step through the keywords and place the document in the category of the first keyword that matches. Since we provide only a few keywords for each class, classification by keyword matching is both inaccurate and incomplete. Keywords tend to provide high-precision and low-recall; this brittleness will leave many documents unlabeled. Some documents will match keywords from the wrong class. In general we expect the low recall of the keywords to be the dominating factor in overall error. In our experimental domain, for example, 59% of the unlabeled documents do not contain any keywords.

### 5.2. The Bootstrapping Iterations

The goal of the bootstrapping iterations is to generate a naive Bayes classifier from seed information and the inputs: the (inaccurate and incomplete) preliminary labels, the unlabeled data and the class hierarchy. Many bootstrapping algorithms assign labels to the unlabeled data, and then choose just a few of these to incorporate into training at each step. In our algorithm, we take a different approach. At each bootstrapping step we assign *probabilistic* labels to *all* the unlabeled data, and incorporate the entire set into training. Expectation-Maximization

Table VII. An outline of the bootstrapping algorithm described in Sections 5.1 and 5.2.

- **Inputs:** A collection of unlabeled training documents, a class hierarchy, and a few keywords for each class.
- Generate preliminary labels for as many of the unlabeled documents as possible by term-matching with the keywords in a rule-list fashion.
- Initialize all the $\lambda_j$'s to be uniform along each path from a leaf class to the root of the class hierarchy.
- Iterate the EM algorithm:
    - **(M-step)** Build the maximum likelihood multinomial at each node in the hierarchy given the class probability estimates for each document (Equations 10 and 11). Normalize all the $\lambda_j$'s along each path from a leaf class to the root of the class hierarchy so that they sum to 1.
    - **(E-step)** Calculate the expectation of the class labels of each document using the classifier created in the M-step (Equation 12). Increment the new $\lambda_j$'s by attributing each word of held-out data probabilistically to the ancestors of each class.
- **Output:** A naive Bayes classifier that takes an unlabeled test document and predicts a class label.

is the bootstrapping process we use to iteratively estimate these probabilistic labels and the parameters of the naive Bayes classifier. We begin a detailed description of the bootstrapping iteration with a short overview of supervised naive Bayes text classification, then proceed to explain EM as a bootstrapping process, and conclude by presenting hierarchical shrinkage, an augmentation to basic EM estimation that uses the class hierarchy.

#### 5.2.1. *The naive Bayes framework*

We build on the framework of multinomial naive Bayes text classification (Lewis, 1998; McCallum & Nigam, 1998). It is useful to think of naive Bayes as estimating the parameters of a probabilistic generative model for text documents. In this model, first the class of the document is selected. The words of the document are then generated based on the parameters of a class-specific multinomial (*i.e.* unigram model). Thus, the classifier parameters consist of the class prior probabilities and the class-conditioned word probabilities. Each class, $c_j$, has a document frequency relative to all other classes, written $\mathrm{P}(c_j)$. For every word $w_t$ in the vocabulary $V$, $\mathrm{P}(w_t|c_j)$ indicates the frequency that the classifier expects word $w_t$ to occur in documents in class $c_j$.

In the standard supervised setting, learning of the parameters is accomplished using a set of labeled training documents, $\mathcal{D}$. To estimate the word probability parameters, $\mathrm{P}(w_t|c_j)$, we count the frequency with which word $w_t$ occurs among all word occurrences for documents in class $c_j$. We supplement this with Laplace smoothing that primes each estimate with a count of one to avoid probabilities of zero. Let $N(w_t, d_i)$ be the count of the number of times word $w_t$ occurs in document $d_i$, and define $\mathrm{P}(c_j|d_i) \in \{0, 1\}$, as given by the document's class label. Then, the estimate of the probability of word $w_t$ in class $c_j$ is:

$$\mathrm{P}(w_t|c_j) = \frac{1 + \sum_{d_i \in \mathcal{D}} N(w_t, d_i)\mathrm{P}(c_j|d_i)}{|V| + \sum_{s=1}^{|V|} \sum_{d_i \in \mathcal{D}} N(w_s, d_i)\mathrm{P}(c_j|d_i)}. \tag{10}$$

The class prior probability parameters are set in the same way, where $|\mathcal{C}|$ indicates the number of classes:

$$\mathrm{P}(c_j) = \frac{1 + \sum_{d_i \in \mathcal{D}} \mathrm{P}(c_j|d_i)}{|\mathcal{C}| + |\mathcal{D}|}. \tag{11}$$

Given an unlabeled document and a classifier, we determine the probability that the document belongs in class $c_j$ using Bayes' rule and the naive Bayes assumption—that the words in a document occur independently of each other given the class. If we denote $w_{d_{i,k}}$ to be the $k$th word in document $d_i$, then classification becomes:

$$\begin{aligned} \mathrm{P}(c_j|d_i) &\propto \mathrm{P}(c_j)\mathrm{P}(d_i|c_j) \\ &\propto \mathrm{P}(c_j) \prod_{k=1}^{|d_i|} \mathrm{P}(w_{d_{i,k}}|c_j). \end{aligned} \tag{12}$$

Empirically, when given a large number of training documents, naive Bayes does a good job of classifying text documents (Lewis, 1998). More complete presentations of naive Bayes for text classification are provided by Mitchell (1997) and McCallum and Nigam (1998).

5.2.2. *Parameter estimation from unlabeled data with EM*

In a standard supervised setting, each document comes with a label. In our bootstrapping scenario, the documents are unlabeled, except for the preliminary labels from keyword matching that are incomplete and not completely correct. In order to estimate the parameters of a naive Bayes classifier using all the documents, we use EM to generate probabilistically-weighted class labels. This results in classifier parameters that are more likely given all the data.

EM is a class of iterative algorithms for maximum likelihood or maximum a posteriori parameter estimation in problems with incomplete data (Dempster, Laird, & Rubin, 1977). Given a model of data generation, and data with some missing values, EM iteratively uses the current model to estimate the missing values, and then uses the missing value estimates to improve the model. Using all the available data, EM will locally maximize the likelihood of the parameters and give estimates for the missing values. In our scenario, the class labels of the documents are the missing values.

In implementation, using EM for bootstrapping is an iterative two-step process. Initially, the parameter estimates are set in the standard naive Bayes way from just the preliminarily labeled documents. Then we iterate the E- and M-steps. The E-step calculates probabilistically-weighted class labels, $\mathrm{P}(c_j|d_i)$, for every document using the classifier and Equation 12. The M-step estimates new classifier parameters using all the documents, by Equations 10 and 11, where $\mathrm{P}(c_j|d_i)$ is now continuous, as given by the E-step. We iterate the E- and M-steps until the classifier converges. The initialization step from the preliminary labels identifies a starting point for EM to find a good local maxima for the classification task.

In previous work (Nigam, McCallum, Thrun, & Mitchell, 2000), we have shown this bootstrapping technique significantly increases text classification accuracy when given limited amounts of labeled data and large amounts of unlabeled data. Here, we use the preliminary labels to provide the starting point for EM. The EM iterations both correct the preliminary labels and complete the labeling for the remaining documents.

#### 5.2.3. *Improving sparse data estimates with shrinkage*

Even when provided with a large pool of documents, naive Bayes parameter estimation during bootstrapping will suffer from sparse data problems because there are so many parameters to estimate ($|V||C| + |C|$). Fortunately we can further alleviate the sparse data problem by leveraging the class hierarchy with a statistical technique called *shrinkage*.

Consider trying to estimate the probability of the word "intelligence" in the class NLP. This word should clearly have non-negligible probability there; however, with limited training data we may be unlucky, and the observed frequency of "intelligence" in NLP may be very far from its true expected value. One level up the hierarchy, however, the Artificial Intelligence class contains many more documents (the union of all the children). There, the probability of the word "intelligence" can be more reliably estimated.

Shrinkage calculates new word probability estimates for each leaf class by a *weighted average* of the estimates on the path from the leaf to the root. The technique balances a trade-off between specificity and reliability. Estimates in the leaf are most specific but unreliable; further up the hierarchy estimates are more reliable but unspecific. We can calculate mixture weights for the averaging that are guaranteed to maximize the likelihood of held-out data with the EM algorithm during bootstrapping.

One can think of hierarchical shrinkage as a generative model that is slightly augmented from the one described in Section 5.2.1. As before, a class (leaf) is selected first. Then, for each word occurrence in the document, an ancestor of the class (including itself) is selected according to the shrinkage weights. Then, the word itself is chosen based on the multinomial word distribution of that ancestor. If each word in the training data were labeled with which ancestor was responsible for generating it, then estimating the mixture weights would be a simple matter of maximum likelihood estimation from the ancestor emission counts. But these ancestor labels are not provided in the training data, and hence we use EM to fill in these missing values. During EM, we estimate these vertical mixture weights concurrently with the class word probabilities.

More formally, let $\{\mathrm{P}^1(w_t|c_j), \ldots, \mathrm{P}^k(w_t|c_j)\}$ be word probability estimates, where $\mathrm{P}^1(w_t|c_j)$ is the maximum likelihood estimate using training data just in the leaf, $\mathrm{P}^2(w_t|c_j)$ is the maximum likelihood estimate in the parent using the training data from the union of the parent's children, $\mathrm{P}^{k-1}(w_t|c_j)$ is the estimate at the root using all the training data, and $\mathrm{P}^k(w_t|c_j)$ is the uniform estimate ($\mathrm{P}^k(w_t|c_j) = 1/|V|$). The interpolation weights among $c_j$'s "ancestors" (which we define to include $c_j$ itself) are written $\{\lambda_j^1, \lambda_j^2, \ldots, \lambda_j^k\}$, where $\sum_{a=1}^k \lambda_j^a = 1$. The new word probability estimate based on shrinkage, denoted $\check{\mathrm{P}}(w_t|c_j)$, is then

$$\check{\mathrm{P}}(w_t|c_j) = \lambda_j^1 \mathrm{P}^1(w_t|c_j) + \ldots + \lambda_j^k \mathrm{P}^k(w_t|c_j). \qquad (13)$$

The $\lambda_j$ vectors are calculated by the iterations of EM. In the E-step we calculate for each class $c_j$ and each word of unlabeled held-out data $\mathcal{H}$, the probability that the word was generated by the $i$th ancestor. In the M-step, we normalize the sum of these expectations to obtain new mixture weights $\lambda_j$. The held-out documents are chosen randomly from the training set. Without the use of held-out data, all the mixture weights would concentrate in the leaves, since the most-specific model would best fit the training data. EM still converges with this use of held-out data; in fact, the likelihood surface is convex, and hence it is guaranteed to converge to the global maximum.

Specifically, we begin by initializing the $\lambda$ mixture weights along each path from a leaf to a uniform distribution. Let $\beta_j^a(w_{d_{i,k}})$ denote the probability that the $a$th ancestor of $c_j$ was used to generate word occurrence $w_{d_{i,k}}$. The E-step consists of estimating the $\beta$'s:

$$\beta_j^a(w_{d_{i,k}}) = \frac{\lambda_j^a \mathrm{P}^a(w_{d_{i,k}}|c_j)}{\sum_m \lambda_j^m \mathrm{P}^m(w_{d_{i,k}}|c_j)}. \tag{14}$$

In the M-step, we derive new and guaranteed improved weights, $\lambda$, by summing and normalizing the $\beta$'s:

$$\lambda_j^a = \frac{\sum_{w_{d_{i,k}} \in \mathcal{H}} \beta_j^a(w_{d_{i,k}}) \mathrm{P}(c_j|d_i)}{\sum_b \sum_{w_{d_{i,k}} \in \mathcal{H}} \beta_j^b(w_{d_{i,k}}) \mathrm{P}(c_j|d_i)}. \tag{15}$$

The E- and M-steps iterate until the $\lambda$'s converge. These weights are then used to calculate new shrinkage-based word probability estimates, as in Equation 13. Classification of new test documents is performed just as before (Equation 12), where the Laplace estimates of the word probability estimates are replaced by shrinkage-based estimates.

A more complete description of hierarchical shrinkage for text classification is presented by McCallum et al. (1998).

### 5.3. Experimental Results

In this section, we provide empirical evidence that bootstrapping a text classifier from unlabeled data can produce a high-accuracy text classifier. As a test domain, we use computer science research papers. We have created a 70-leaf hierarchy of computer science topics, part of which is shown in Figure 10. Creating the hierarchy took about 60 minutes, during which we examined conference proceedings, and explored computer science sites on the Web. Selecting a few keywords associated with each node took about 90 minutes. A test set was created by expert hand-labeling of a random sample of 625 research papers from the 30,682 papers in the Cora archive at the time we began these experiments. Of these, 225 (about one-third) did not fit into any category, and were discarded—resulting in a 400 document test set. Labeling these documents took about six hours. Some of the discarded papers were outside the area of computer science (*e.g.* astrophysics papers), but most of these were papers that with a more complete hierarchy would be considered computer science papers. The class frequencies of the data are skewed, but not drastically; on the test set, the most populous class accounted for only 7% of the documents.

Table VIII. Classification results with different techniques: keyword matching, naive Bayes, Bootstrapping and Human agreement. The classification accuracy, and the number of labeled, keyword-matched preliminarily-labeled (P-Labeled), and unlabeled documents used by each variant are shown.

| Method | # Labeled | # P-Labeled | # Unlabeled | Accuracy |
|---|---|---|---|---|
| Keyword Matching | — | — | — | 46% |
| Naive Bayes | 100 | — | — | 30% |
| Naive Bayes | 399 | — | — | 47% |
| Naive Bayes | — | 12,657 | — | 47% |
| Bootstrapping | — | 12,657 | — | 63% |
| Bootstrapping | — | 12,657 | 18,025 | 66% |
| Human Agreement | — | — | — | 72% |

Each research paper is represented as the words of the title, author, institution, references, and abstract. A detailed description of how these segments are automatically extracted is provided in Section 4. Words occurring in fewer than five documents and words on a standard stoplist were discarded. No stemming was used. Bootstrapping was performed using the algorithm outlined in Table VII.

Table VIII shows results with different classification techniques used. The rule-list classifier based on the keywords alone provides 46% accuracy.[5] As an interesting time comparison, about 100 documents could have been labeled in the time it took to generate the keyword lists. Naive Bayes accuracy with 100 labeled documents is only 30%. It takes about four times as much labeled training data to provide comparable accuracy to simple keyword matching; with 399 labeled documents (using our test set in a leave-one-out-fashion), naive Bayes reaches 47%. This result alone shows that hand-labeling sets of data for supervised learning can be expensive in comparison to alternate techniques.

When running the bootstrapping algorithm, 12,657 documents are given preliminary labels by keyword matching. EM and shrinkage incorporate the remaining 18,025 documents, "fix" the preliminary labels and leverage the hierarchy; the resulting accuracy is 66%. As an interesting comparison, agreement on the test set between two human experts was 72%. These results show that our bootstrapping algorithm can generate competitive classifications without the use of large hand-labeled sets of data.

[5] The 43% of documents in the test set containing no keywords are not assigned a class by the rule-list classifier, and are assigned the most populous class by default.

A few further experiments reveal some of the inner-workings of bootstrapping. If we build a naive Bayes classifier in the standard supervised way from the 12,657 preliminarily labeled documents the classifier gets 47% accuracy. This corresponds to the performance for the first iteration of bootstrapping. Note that this matches the accuracy of traditional naive Bayes with 399 labeled training documents, but that it requires less than a quarter the human labeling effort. If we run bootstrapping without the 18,025 documents left unlabeled by keyword matching, accuracy reaches 63%. This indicates that shrinkage and EM on the preliminarily labeled documents is providing substantially more benefit than the remaining unlabeled documents.

### 5.4. Discussion

One explanation for the small impact of the 18,025 documents left unlabeled by keyword matching is that many of these do not fall naturally into the hierarchy. Remember that about one-third of the 30,000 documents fall outside the hierarchy. Most of these will not be given preliminary labels by keyword matching. The presence of these outlier documents skews EM parameter estimation. A more inclusive computer science hierarchy would allow the unlabeled documents to benefit classification more.

However, even without a complete hierarchy, we could use these documents if we could identify these outliers. Some techniques for robust estimation with EM are discussed by McLachlan and Basford (1988). One specific technique for these text hierarchies is to add extra leaf nodes containing uniform word distributions to each interior node of the hierarchy in order to capture documents not belonging in any of the predefined topic leaves. This should allow EM to perform well even when a large percentage of the documents do not fall into the given classification hierarchy. A similar approach is also planned for research in topic detection and tracking (TDT) (Baker, Hofmann, McCallum, & Yang, 1999). Experimentation with these techniques is an area of ongoing research.

In other future work we will investigate different ways of initializing bootstrapping, with keywords and otherwise. We plan to refine our probabilistic model to allow for documents to be placed in interior hierarchy nodes, documents to have multiple class assignments, and classes to be modeled with multiple mixture components. We are also investigating principled methods of re-weighting the word features for "semi-supervised" clustering that will provide better discriminative training with unlabeled data.

Here, we have shown the application of our bootstrapping process to populating a hierarchy for Cora. Topic hierarchies are often an integral part of most portals, although they are typically hand-built and maintained. The techniques demonstrated here are generally applicable to any topic hierarchy, and should become a powerful tool for populating topic hierarchies with a minimum of human effort.

## 6. Related Work

Several related research projects investigate the gathering and organization of specialized information on the Internet. The WebKB project (Craven, DiPasquo, Freitag, McCallum, Mitchell, Nigam, & Slattery, 1998) focuses on the collection and organization of information from the Web into knowledge bases. This project also has a strong emphasis on using machine learning techniques, including text classification and information extraction, to promote easy re-use across domains. Two example domains, computer science departments and companies, have been developed.

The CiteSeer project (Lawrence, Giles, & Bollacker, 1999) has also developed a search engine for computer science research papers. It provides similar functionality for searching and linking of research papers. They locate papers by querying search engines with paper-indicative words. Information is extracted from paper headers and references by using an invariants first ordering of heuristics. They provide a hierarchy of computer science with hubs and authorities rankings on the papers. They provide similarity rankings between research papers based on words and citations. CiteSeer focuses on the domain of research papers, and has particularly strong features for autonomous citation indexing and the viewing of the textual context in which a citation was made.

The New Zealand Digital Library project (Witten, Nevill-Manning, McNab, & Cunnningham, 1998) has created publicly-available search engines for domains from computer science technical reports to song melodies. The emphasis of this project is on the creation of full-text searchable digital libraries, and not on machine learning techniques that can be used to autonomously generate such repositories. The web sources for their libraries are manually identified. No high-level organization of the information is given. No information extraction is performed and, for the paper repositories, no citation linking is provided.

The WHIRL project (Cohen, 1998) is an effort to integrate a variety of topic-specific sources into a single domain-specific search engine. Two demonstration domains of computer games and North American birds integrate information from many sources. The emphasis is on providing soft matching for information retrieval searching. Information is extracted from web pages by hand-written extraction patterns that are customized for each web source. Recent WHIRL research (Cohen & Fan, 1999) learns general wrapper extractors from examples.

## 7. Conclusions and Future Work

The amount of information available on the Internet continues to grow exponentially. As this trend continues, we argue that not only will the public need powerful tools to help them sort through this information, but the *creators* of these tools will need intelligent techniques to help them build and maintain these services. This paper has shown that machine learning techniques can significantly aid the creation and maintenance of domain-specific portals and search engines. We have presented new research in reinforcement learning, text classification and information extraction towards this end.

In addition to the future work discussed above, we also see many other areas where machine learning can further automate the construction and maintenance of portals such as ours. For example, text classification can decide which documents on the Web are relevant to the domain. Unsupervised clustering can automatically create a topic hierarchy and generate keywords (Hofmann & Puzicha, 1998; Baker et al., 1999). Citation graph analysis can identify seminal papers (Kleinberg, 1999; Chang et al., 1999). We anticipate developing a suite of many machine learning techniques so that the creation of portals can be accomplished quickly and easily.

## Acknowledgements

Most of the work in this paper was performed while all the authors were at Just Research. Kamal Nigam was supported in part by the DARPA HPKB program under contract F30602-97-1-0215.

## References

Baker, D., Hofmann, T., McCallum, A., & Yang, Y. (1999). A hierarchical probabilistic model for novelty detection in text. Tech. rep., Just Research. http://www.cs.cmu.edu/~mccallum.

Baum, L. E. (1972). An inequality and associated maximization technique in statistical estimation of probabilistic functions of a Markov process. *Inequalities*, *3*, 1–8.

Bellman, R. E. (1957). *Dynamic Programming.* Princeton University Press, Princeton, NJ.

Bikel, D. M., Miller, S., Schwartz, R., & Weischedel, R. (1997). Nymble: a high-performance learning name-finder. In *Procedings of the Fifth Conference on Applied Natural Language Processing (ANLP-97)*, pp. 194–201.

Blum, A., & Mitchell, T. (1998). Combining labeled and unlabeled data with co-training. In *Proceedings of the 11th Annual Conference on Computational Learning Theory (COLT '98)*, pp. 92–100.

Boyan, J., Freitag, D., & Joachims, T. (1996). A machine learning architecture for optimizing web search engines. In *AAAI-96 Workshop on Internet-Based Information Systems.*

Chakrabarti, S., van der Berg, M., & Dom, B. (1999). Focused crawling: a new approach to topic-specific Web resource discovery. In *Proceedings of 8th International World Wide Web Conference (WWW8).*

Chang, H., Cohn, D., & McCallum, A. (1999). Creating customized authority lists. http://www.cs.cmu.edu/∼mccallum.

Chen, S. F., & Goodman, J. T. (1998). An empirical study of smoothing techniques for language modeling. Tech. rep. TR-10-98, Computer Science Group, Harvard University.

Cho, J., Garcia-Molina, H., & Page, L. (1998). Efficient crawling through URL ordering. In *Proceedings of the Seventh World-Wide Web Conference (WWW7).*

Cohen, W., & Fan, W. (1999). Learning page-independent heuristics for extracting data from web pages. In *AAAI Spring Symposium on Intelligent Agents in Cyberspace.*

Cohen, W. (1998). A web-based information system that reasons with structured collections of text. In *Proceedings of the Second International Conference on Autonomous Agents (Agents '98)*, pp. 400–407.

Craven, M., DiPasquo, D., Freitag, D., McCallum, A., Mitchell, T., Nigam, K., & Slattery, S. (1998). Learning to extract symbolic knowledge from the World Wide Web. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pp. 509–516.

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

Freitag, D., & McCallum, A. (1999). Information extraction with HMMs and shrinkage. In *Proceedings of the AAAI-99 Workshop on Machine Learning for Information Extraction.*

Giles, C. L., Bollacker, K. D., & Lawrence, S. (1998). CiteSeer: An autonomous citation indexing system. In *Digital Libraries 98 - Third ACM Conference on Digital Libraries*, pp. 89–98.

Hofmann, T., & Puzicha, J. (1998). Statistical models for co-occurrence data. Tech. rep. AI Memo 1625, Artificial Intelligence Laboratory, MIT.

Joachims, T., Freitag, D., & Mitchell, T. (1997). Webwatcher: A tour guide for the World Wide Web. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, pp. 770–777.

Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, *4*, 237–285.

Kearns, M., Mansour, Y., & Ng, A. (2000). Approximate planning in large POMDPs via reusable trajectories. In *Advances in Neural Information Processing Systems 12*. The MIT Press.

Kleinberg, J. (1999). Authoritative sources in a hyperlinked environment. *Journal of the ACM*, *46*.

Kupiec, J. (1992). Robust part-of-speech tagging using a hidden Markov model. *Computer Speech and Language*, *6*, 225–242.

Lawrence, S., Giles, C. L., & Bollacker, K. (1999). Digital libraries and autonomous citation indexing. *IEEE Computer*, *32*(6), 67–71.

Leek, T. R. (1997). Information extraction using hidden Markov models. Master's thesis, UC San Diego.

Lewis, D. D. (1998). Naive (Bayes) at forty: The independence assumption in information retrieval. In *Machine Learning: ECML-98, Tenth European Conference on Machine Learning*, pp. 4–15.

McCallum, A., & Nigam, K. (1998). A comparison of event models for naive Bayes text classification. In *AAAI-98 Workshop on Learning for Text Categorization*. http://www.cs.cmu.edu/~mccallum.

McCallum, A., Rosenfeld, R., Mitchell, T., & Ng, A. (1998). Improving text clasification by shrinkage in a hierarchy of classes. In *Machine Learning: Proceedings of the Fifteenth International Conference (ICML '98)*, pp. 359–367.

McLachlan, G., & Basford, K. (1988). *Mixture Models*. Marcel Dekker, New York.

Menczer, F. (1997). ARACHNID: Adaptive retrieval agents choosing heuristic neighborhoods for information discovery. In *Machine Learning: Proceedings of the Fourteenth International Conference (ICML '97)*, pp. 227–235.

Merialdo, B. (1994). Tagging english text with a probabilistic model. *Computational Linguistics*, *20*(2), 155–171.

Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill, New York.

Ney, H., Essen, U., & Kneser, R. (1994). On structuring probabilistic dependencies in stochastic language modeling. *Computer Speech and Language*, *8*(1), 1–38.

Nigam, K., McCallum, A., Thrun, S., & Mitchell, T. (2000). Text classification from labeled and unlabeled documents using EM. *Machine Learning*, *39*.

Rabiner, L. R. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, *77*(2), 257–286.

Riloff, E., & Jones, R. (1999). Learning dictionaries for information extraction using multi-level boot-strapping. In *Proceedings of the Sixteenth National Conference on Artificial Intellligence (AAAI-99)*, pp. 474–479.

Stolcke, A., Shriberg, E., Bates, R., Coccaro, N., Jurafsky, D., Martin, R., Meteer, M., Ries, K., Taylor, P., & Ess-Dykema, C. V. (1998). Dialog act modeling for conversational speech. In *AAAI Spring Symposium on Applying Machine Learning to Discourse Processing*, pp. 98–105.

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

Tesauro, G., & Galperin, G. R. (1997). On-line policy improvement using monte-carlo search. In *Advances in Neural Information Processing Systems 9*, pp. 1068–1074. The MIT Press.

Torgo, L., & Gama, J. (1997). Regression using classification algorithms. *Intelligent Data Analysis*, *1*(4), 275–292.

Viterbi, A. J. (1967). Error bounds for convolutional codes and an asymtotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, *IT-13*, 260–269.

Witten, I. H., Nevill-Manning, C., McNab, R., & Cunnningham, S. J. (1998). A public digital library based on full-text retrieval: Collections and experience. *Communications of the ACM*, *41*(4), 71–75.

Yamron, J., Carp, I., Gillick, L., Lowe, S., & van Mulbregt, P. (1998). A hidden Markov model approach to text segmentation and event tracking. In *Procedings of International Conference on Acoustics, Speech and Signal Processing (ICASSP-98)* Seattle, Washington.

Yarowsky, D. (1995). Unsupervised word sense disambiguation rivaling supervised methods. In *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics (ACL-95)*, pp. 189–196.