# Building Domain-Specific Search Engines with Machine Learning Techniques

**Andrew McCallum**[‡†] mccallum@justresearch.com
**Kamal Nigam**[†] knigam@cs.cmu.edu
**Jason Rennie**[†] jr6b@andrew.cmu.edu
**Kristie Seymore**[†] kseymore@ri.cmu.edu

[‡]Just Research
4616 Henry Street
Pittsburgh, PA 15213

[†]School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

**Abstract**

Domain-specific search engines are growing in popularity because they offer increased accuracy and extra functionality not possible with the general, Web-wide search engines. For example, *www.campsearch.com* allows complex queries by age-group, size, location and cost over summer camps. Unfortunately these domain-specific search engines are difficult and time-consuming to maintain. This paper proposes the use of machine learning techniques to greatly automate the creation and maintenance of domain-specific search engines. We describe new research in reinforcement learning, information extraction and text classification that enables efficient spidering, identifying informative text segments, and populating topic hierarchies. Using these techniques, we have built a demonstration system: a search engine for computer science research papers. It already contains over 50,000 papers and is publicly available at *www.cora.justresearch.com*.

## 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 HotBot offer high coverage, they often provide only low precision, even for detailed queries.

When we know that we want information of a certain type, or on a certain topic, a domain-specific search engine can be a powerful tool. For example:

- www.campsearch.com allows the user to search for summer camps for children and adults. The user can query the system based on geographic location, cost, duration and other requirements.
- www.netpart.com lets the user search over company pages by hostname, company name, and location.
- 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.
- www.maths.usyd.edu.au/MathSearch.html lets the user search web pages about mathematics.
- 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, domain-specific search engines are becoming increasingly popular. Unfortunately, however, building these search engines is a labor-intensive process, typically requiring significant and ongoing human effort.

This paper describes the *Ra Project*—an effort to automate many aspects of creating and maintaining domain-specific search engines by using machine learning techniques. These techniques allow search engines to be created quickly with minimal effort and are suited for re-use across many domains. This paper presents machine learning methods for spidering in an efficient topic-directed manner, extracting topic-relevant substrings, and building a browseable topic hierarchy. These approaches are briefly described in the following three paragraphs.

Every search engine 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 search engine. When aiming to populate a domain-specific search engine, the spider need not explore the Web indiscriminantly, but should explore in a directed fashion in order to find domain-relevant documents efficiently. We frame the spidering task in a reinforcement learning framework (Kaelbling, Littman, & Moore 1996), allowing us to precisely and mathematically define "optimal behavior." This approach provides guidance for designing an intelligent spider that aims to select hyperlinks optimally. Our

experimental results show that a reinforcement learning spider is 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 search engine allows the user to search over these features in a way that general search engines cannot. Information extraction, the process of automatically finding specific 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). Our initial algorithm extracts fields such as the title, authors, institution, and journal name from research paper reference sections 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) that succeed in this task without requiring large sets of labeled training data. The extensions reduce the need for human effort in training the classifier by (1) using keyword matching to automatically assign approximate labels, (2) using a statistical technique called *shrinkage* that finds more robust parameter estimates by taking advantage of the hierarchy, and (3) increasing accuracy further by iterating Expectation-Maximization to probabilistically reassign approximate labels and incorporate unlabeled data. Use of the resulting algorithms places documents into a 70-leaf computer science hierarchy with 66% accuracy—performance approaching human agreement levels.

## 2 The Cora Search Engine

We have brought all the above-described machine learning techniques together in a demonstration system: a domain-specific search engine on computer science research papers named 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, and provides bibliographic information about each paper. Our hope is that in addition to providing 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 the Computing Research Repository (*xxx.lanl.gov/archive/cs*), by providing functionality and coverage not available online elsewhere.

The construction of a search engine can be decomposed into three functional stages: collecting new information, collating and extracting from that information, and presenting it in a publicly-available web interface. Cora implements each stage by drawing upon machine learning techniques described in this paper.

The first stage is the collection of computer science research papers. A spider crawls the Web, starting from the home pages of computer science departments and laboratories. Using reinforcement learning, it efficiently explores the Web, collecting all postscript documents it finds. Nearly all computer science papers are in postscript format, though we are adding more formats, such as PDF. These postscript documents are then converted into plain text. If the document can be reliably determined to have the format of a research paper (*e.g.* by having Abstract and Reference sections), 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 second stage of building a search engine is to extract relevant knowledge from each paper. To this end, the beginning of each paper (up to the abstract) is passed through an information extraction system that automatically finds the title, author, institution and other important header information. Additionally, the bibliography section of each paper is located, individual references identified, and each reference broken down into the appropriate fields, such as author, title, journal, and date. Using this extracted information, reference and paper matches are made—grouping citations to the same paper together, and matching citations to papers in Cora. Of course, many papers that are cited do not appear in the repository. This matching procedure is similar to one described by Bollacker, Lawrence, & Giles (1998), except that we use additional field-level constraints provided by knowing, for example, the title and authors of each paper.

The third stage is to provide a publicly-available user interface. We have implemented two methods for finding papers. First, a search engine over all the papers is provided. It supports commonly-used searching syntax for queries, including `+`, `-`, and phrase searching with `""`, and ranks resulting matches by the weighted log of term frequency, summed over all query terms. It also allows searches restricted to extracted fields, such as authors and titles. Query response time is usually less than a second. The results of search queries are presented as in Figure 1. Additionally, each individual paper has a "details" page that shows all the relevant information, 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

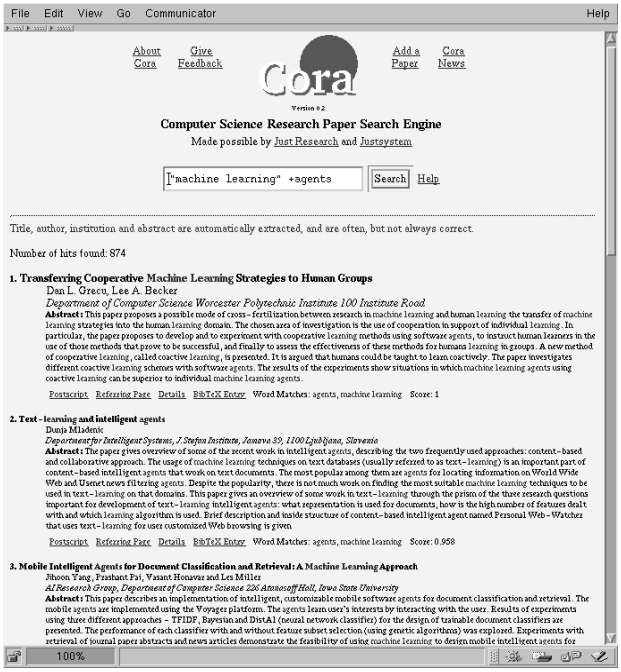


Figure 1: A screen shot of the query results page of the Cora search engine (*www.cora.justresearch.com*). Extracted paper titles, authors and abstracts are provided at this level.

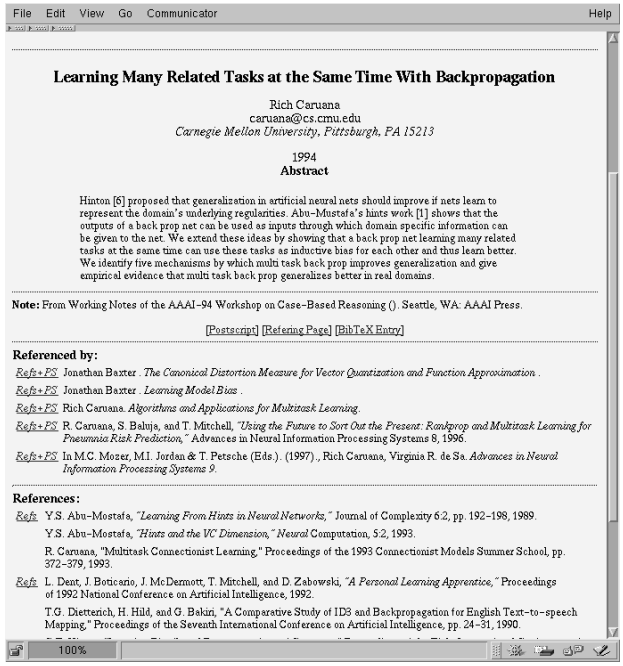


Figure 2: A screen shot of a details page of the Cora search engine. At this level, all extracted information about a paper is displayed, including the citation linking, which are hyperlinks to other details pages.

of this is shown in Figure 2. We also provide automatically constructed BibTeX entries, general Cora information links, and a mechanism for submitting new papers and web sites for spidering.

The other user interface access method is through a topic hierarchy, similar to that provided by Yahoo, but customized specifically for computer science research. 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 node. By following hyperlinks to traverse the topic hierarchy, the most-cited papers in each research topic can be found.

## 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 search engine. Extensive spidering is the key to obtaining high coverage by the major Web search engines, such as AltaVista and HotBot. Since the goal of these general-purpose search engines is to provide search capabilities over the Web as a whole, for the most part they simply 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 search engine, 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 many 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 to perform efficient spidering.

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, & Page (1998) suggest a number of heuristic ordering metrics for choosing which link to crawl next when searching for certain categories of web pages. Additionally, there are systems that use reinforcement learning for non-spidering Web tasks. WebWatcher (Joachims, Freitag, & Mitchell 1997) is a browsing assistant that uses a combination of supervised and reinforcement learning to help a user find information by recommending which hyperlinks to follow. Laser uses reinforcement learning to tune the parameters of

a search engine (Boyan, Freitag, & Joachims 1996).

### 3.1 Reinforcement Learning

In machine learning, the term "reinforcement learning" refers to a framework for learning optimal decision making from rewards or punishment (Kaelbling, Littman, & Moore 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."

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, $T : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$, and a reward function, $R : \mathcal{S} \times \mathcal{A} \rightarrow \Re$. At each time step, the learner (also called the *agent*) selects an action, and then as a result is given a reward and its 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. A *discount factor*, $\gamma$, $0 \leq \gamma < 1$, expresses "inflation," making sooner rewards more valuable than later rewards. Accordingly, when following policy $\pi$, we can define the *value* of each state to be:

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

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, $V^{\pi}(s)$, for 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:

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

We can now define the optimal policy in terms of $Q$ 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 agent's actions are moving among the grid squares of the maze. The agent receives a reward for finding each piece of cheese. The state is 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 agent only receives immediate reward for finding a maze square containing cheese, but that in order to act optimally it must choose actions considering 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 bit vector indicating which on-topic documents remain to be consumed. The state does not include the current "position" of the agent since a crawler can go to any URL next. The number of actions is large and dynamic, in that it depends on which documents the spider has visited so far.

The key features of topic-specific spidering that make reinforcement learning the proper framework for defining the optimal solution are: (1) performance is measured in terms of reward over time, and (2) the environment presents situations with delayed reward.

### 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: two to the power of the number of on-topic documents on the Web. The action space is also large: the number of unique URLs with incoming links on the Web. Thus we need to make some simplifying assumptions in order to make the problem tractable and to aid generalization. Note, however, that by defining the exact solution in terms of the optimal policy, and making our assumptions explicit, we will better understand what inaccuracies we have introduced, and how to select areas of future work that will improve performance further. The assumptions we choose initially are the following two: (1) we assume that the state is independent of which on-topic documents have already been consumed; that is, we collapse all states into one, and (2) we assume that the relevant distinctions between the actions can be captured by the words in the neighborhood of the hyperlink corresponding to each action.

Thus our $Q$ function becomes a mapping from a "bag-of-words" to a scalar (sum of future reward). Learning to perform efficient spidering then involves only two remaining sub-problems: (1) gathering training data consisting of bag-of-words/future-reward pairs, and (2) learning a mapping using the training data.

There are several choices for how to gather training data. Although the agent could learn from experience on-line, we currently train the agent off-line, using collections of already-found documents and hyperlinks.

In the vocabulary of traditional reinforcement learning, this means that the state transition function, $T$, and the reward function, $R$, are known, and we learn the $Q$ function by dynamic programming in the original, uncollapsed state space.

We represent the mapping using a collection of naive Bayes text classifiers (see Section 5.2). We perform the 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, place the hyperlinks into the bin corresponding to their $Q$ values by dynamic programming, and use the hyperlinks' neighborhood text as training data for a naive Bayes text classifier. We define a hyperlink's neighborhood to be two bags-of-words: 1) the full text of the page on which the hyperlink is located, and 2) the anchor text of the hyperlink and portions of the URL.[1] For each hyperlink, we calculate the probabilistic class membership of each bin. Then the reward value of a hyperlink is estimated by taking a weighted average of each bins' reward value, using the probabilistic class memberships as weights.

### 3.4 Data and Experimental Results

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. We perform a series of four test/train splits, in which the data from three universities was used to train a spider that then is tested on the fourth. The target pages (for which a reward of 1 is given) are computer science research papers. They are identified with very high precision by the simple hand-coded algorithm mentioned in Section 2.

We present results of two different reinforcement learning spiders and compare them to breadth-first search. Immediate uses $\gamma = 0$, utilizing only immediate reward in its assignment of hyperlink values. This employs a binary classifier that distinguishes links that do or do not point directly to a research paper. Future uses $\gamma = 0.5$ and represents the $Q$-function with a more finely-discriminating 10-bin classifier that makes use of future reward.

Spiders trained in this fashion are evaluated on each test/train split by spidering the test university. Figure 3 plots the number of research papers found over the course of all the pages visited, averaged over all four universities. Notice that at all times during their progress, the reinforcement learning spiders have found more research papers than Breadth-first.

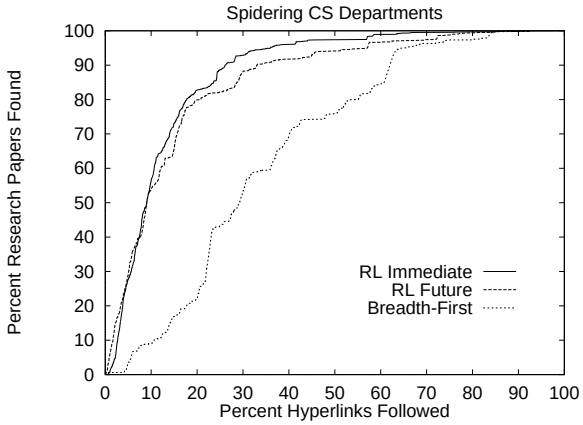


Figure 3: The performance of reinforcement learning spidering versus traditional breadth-first search, averaged over four test/train splits with data from four universities. The reinforcement learning spiders find target documents significantly faster than the traditional method.

One measure of performance is the number of hyperlinks followed before 75% of the research papers are found. Reinforcement learning performs significantly more efficiently, requiring exploration of only 16% of the hyperlinks; in comparison Breadth-first requires 48%. This represents a factor of three increase in spidering efficiency.

Note also that the Future reinforcement learning spider performs better than the Immediate spider in the beginning, when future reward must be used to correctly select among alternative branches, none of which give immediate reward. On average, the Immediate spider takes nearly three times as long as Future to find the first 28 (5%) papers.

In Figure 3, after the first 50% of the papers are found, the Immediate spider performs slightly better than the Future spider. This is because the system has uncovered many links that will give immediate reward if followed, and the Immediate spider recognizes them more accurately. In ongoing work we are investigating techniques for improving classification with the larger number of bins required for regression with future reward. We believe that adding features based on the HTML structure around a hyperlink (headers, titles, and neighboring pages) will improve classification and thus regression.

We are also currently applying the Future spider to other tasks where rewards are more sparse, and thus modeling future reward is more crucial. For example, information about a company's corporate officers is often contained on a single web page in the company's web site; here there is a single reward. Our prelimi-

[1] We have found that performance does not improve when a more restricted set of neighborhood text is chosen.

nary experiments show that our current Future spider performs significantly better than the Immediate spider on these common tasks.

## 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 search engine, the automatic identification of important information can increase the accuracy and efficiency of a directed search.

We use hidden Markov models (HMMs) to extract the fields relevant to research papers, such as title, author, journal and publication date. 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. Our interest in HMMs for information extraction is particularly focused on learning the state and transition structure of the models from training data.

### 4.1 Hidden Markov Models

Hidden Markov models, widely used for speech recognition and part-of-speech tagging (Rabiner 1989; Charniak 1993), provide a natural framework for modeling the production of the headers and reference sections of research papers. 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 $\mathbf{x} = x_1 x_2 \ldots x_l$ 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 $\mathbf{x}$ being emitted by an HMM $M$ is computed as a sum over all possible paths by:

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

where $q_0$ and $q_{l+1}$ are restricted to be $q_I$ and $q_F$ respectively, and $x_{l+1}$ is an end-of-string token. 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 $V(\mathbf{x}|M)$ that has the highest probability of having produced an observation sequence:

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

Fortunately, there is an efficient algorithm, called the Viterbi algorithm (Viterbi 1967), that efficiently recovers this state sequence.

HMMs may be used for information extraction from research papers by formulating a model in the following way: each state is associated with a field class that we want to extract, such as title, author or institution. Each state emits words from a class-specific unigram distribution. We can learn the class-specific unigram distributions and the transition probabilities from data. In our case, we collect BibTeX files from the Web with reference classes explicitly labeled, and use the text from each class as training data for the appropriate unigram model. Transitions between states are estimated directly from a labeled training set, since BibTeX data does not contain this information. In order to label new text with classes, we treat the words from the new text 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.

HMMs have been used in other systems for information extraction and the closely related problems of topic detection and text segmentation. Leek (1997) uses hidden Markov models to extract information about gene names and locations from scientific abstracts. The Nymble system (Bikel *et al.* 1997) deals with named-entity extraction, and a system by Yamron *et al.* (1998) uses an HMM for topic detection and tracking. 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.

### 4.2 Experiments

Our experiments on reference extraction are based on five hundred references that were selected at random from a set of 500 research papers. The words in each of the 500 references were manually tagged with one of the following 13 classes: title, author, institution, location, note, editor, publisher, date, pages, volume, journal, booktitle, and technical report. The tagged references were split into a 300-instance, 6995 word token training set and a 200-instance, 4479 word token test set. Unigram language models were built for each of the thirteen classes from almost 2 million words of BibTeX data acquired from the Web, and were based

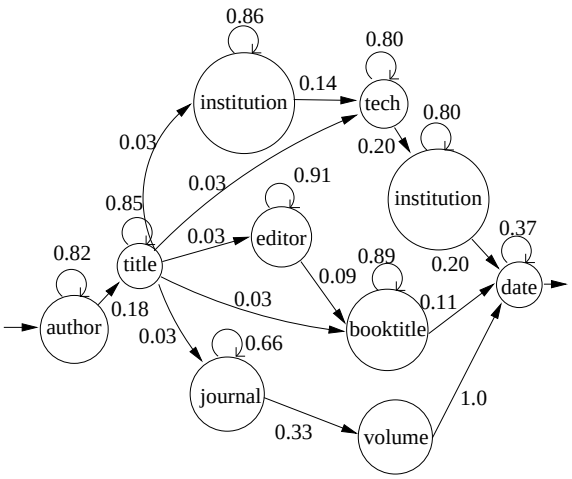


Figure 4: An example HMM built from only five labeled references after merging neighbors and collapsing V-neighbors in the forward and backward directions. Note that the structure is close to many reference section formats.

on a 44,000 word vocabulary. Each HMM state is associated with one class label, and uses the appropriate unigram distribution to provide its emission probabilities. Emission distributions are not re-estimated during the training process.

We examine the potential of learning model structure from data by comparing four different HMMs. The first two models are fully-connected HMMs where each class is represented by a single state. In the first model (HMM-0), the transitions out of each state receive equal probability. Finding the most likely path through this model for an observation sequence is equivalent to consulting each unigram model for each test set word, and setting each word's class to the class of the unigram model that produces the highest probability. The transition probabilities for the second model (HMM-1) are set to the maximum likelihood estimates from the labeled training data. A smoothing count of 1 is added to all transitions to avoid non-zero probabilities.

Next, an HMM is built where each word token in the training set is assigned a single state that only transitions to the state that follows it. Each state is associated with the class label of its word token. From the initial state, there are 300 equiprobable transitions into sequences of states, where each sequence represents the tags for one of the 300 training references. This model consists of 6997 states, and is maximally specific in that its transitions exactly explain the training data.

This HMM is put through a series of state merges in order to generalize the model. First, "neighbor merging" combines all states that share a unique transition and have the same class label. For example, all adjacent title states are merged into one title state, representing the sequence of title words for that reference. 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. After neighbor merging, 1677 states remain in the model (HMM-2).

| Model | # states | Accuracy | |
|---|---|---|---|
| | | Any word | Punc word |
| HMM-0 | 13 | 59.2 | 80.8 |
| HMM-1 | 13 | 91.5 | 92.9 |
| HMM-2 | 1677 | 90.2 | 91.1 |
| HMM-3 | 46 | 91.7 | 92.9 |

Table 1: Word classification accuracy results (%) on 200 test references (4479 words).

Next, the neighbor-merged HMM is put through forward and backward V-merging. For V-merging, any two states that share transitions from or to a common state and have the same label are merged. A simple example of an HMM built from just 5 tagged references after V-merging is shown in Figure 4. Notice that even with just five references, the model closely matches formats found in many reference sections. After V-merging, the HMM is reduced from 1677 states to 46 states (HMM-3).

All four HMM models are used to tag the 200 test references by finding the Viterbi path through each HMM for each reference. The class labels of the states in the Viterbi path are the classifications assigned to each word in the test references. Word classification accuracy results for two testing scenarios are reported in Table 1. In the Any word case, state transitions are allowed to occur after any observation word. In the Punc word case, state transitions to a new state (with a different class label) are only allowed to occur after observations ending in punctuation, since punctuation is often a delimiter between fields in references. For HMM-0, allowing transitions only after words with punctuation greatly increases classification accuracy, since in this case punctuation-delimited phrases are being classified instead of individual words. For the last three cases, the overall classification accuracy is quite high. The V-merged HMM derived directly from the training data (HMM-3) performs at 93% accuracy, as well as the HMM where only one state was allowed per class (HMM-1). For these three cases, limiting state transitions to occur only after words with punctuation improves accuracy by about 1% absolute.

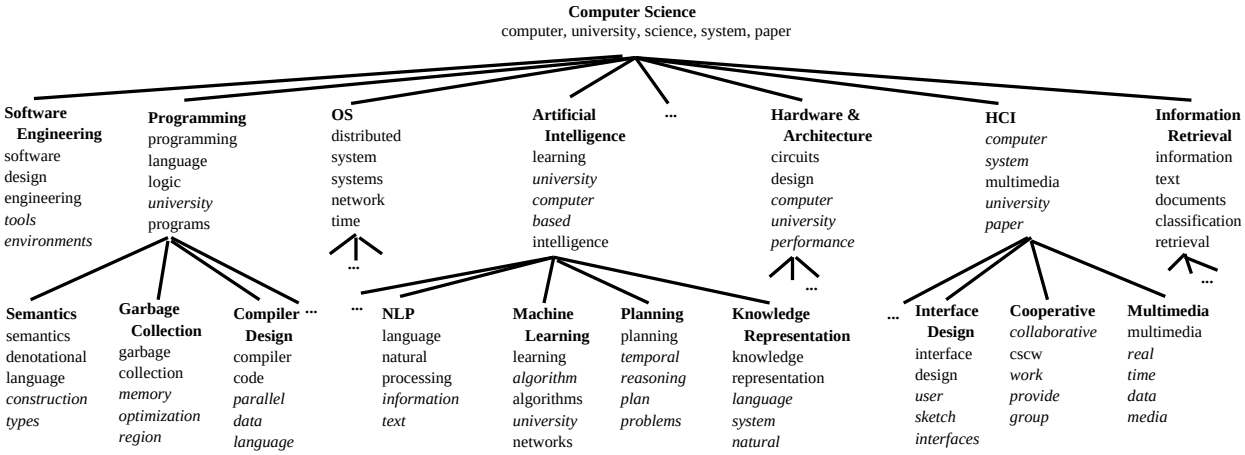


Figure 5: A subset of Cora's topic hierarchy. Each node contains its title, and the five most probable words, as calculated by naive Bayes and shrinkage with vertical word redistribution (Hofmann & Puzicha 1998). Words that were not among the keywords for that class are indicated with italics.

### 4.3 Future Work

All of the experiments presented above use HMMs where the model structure and parameters were estimated directly from labeled training instances. Our future work will focus on using unlabeled training data. Unlabeled training data is preferable to labeled data because generally greater quantities of unlabeled data are available, and model parameters may be more reliably estimated from larger amounts of data. Additionally, manually labeling large amounts of training data is costly and error-prone.

Specifically, if we are willing to fix the model structure, we can use the Baum-Welch estimation technique (Baum 1972) to estimate model parameters. The Baum-Welch method is an Expectation-Maximization procedure for HMMs that finds local likelihood maxima, and is used extensively for acoustic model estimation in automatic speech recognition systems.

We can remove the assumption of a fixed model structure and estimate both model structure and parameters directly from the data using Bayesian Model Merging (Stolcke 1994). Bayesian Model Merging involves starting out with a maximally specific hidden Markov model, where each training observation is represented by a single state. Pairs of states are iteratively merged, generalizing the model until an optimal tradeoff between fit to the training data and a preference for smaller, more generalized models is attained. This merging process can be explained in Bayesian terms by considering that each merging step is looking to find the model that maximizes the posterior probability of the model given the training data. We believe that Bayesian Model Merging, when applied to the V-merged model (HMM-3, 46 states), will result in an intermediate HMM structure that will outperform both the fully-connected model (HMM-1, 13 states) and the V-merged model on the reference extraction task.

We will test both of these induction methods on reference extraction, and will include new experiments on header extraction. We believe that extracting information from headers will be a more challenging problem than references because there is less of an established format for presenting information in the header of a paper.

## 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.

As Yahoo has shown, a topic hierarchy can be a useful, integral part of a search engine. Many search engines (*e.g.* Lycos, Excite, and HotBot) now display hierarchies on their front page. This feature is equally valuable for domain-specific search engines. We have created a 70-leaf hierarchy of computer science topics for Cora, part of which is shown in Figure 5. 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 much more difficult and time-consuming part of creating a hierarchy is populating it with documents that are placed in the correct topic branches. 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 this process with learned text classifiers.

### 5.1 Seeding Naive Bayes using Keywords

One method of classifying documents into a hierarchy is to match them against the keywords 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. If an extensive keyword list is carefully chosen, this method can be reasonably accurate. However, finding enough keywords to obtain broad coverage and finding sufficiently specific keywords to obtain high accuracy can be very difficult; it requires intimate knowledge of the data and a lot of trial and error. Without this extensive effort, keyword matching will be brittle, incapable of finding documents that do not contain matches for selected keywords.

A less brittle approach is provided by naive Bayes, an established text classification algorithm (Lewis 1998; McCallum & Nigam 1998) based on Bayesian machine learning techniques. However, it requires large amounts of labeled training data to work well. Traditionally, training data is labeled by a human, and is difficult and tedious to obtain.

In this paper, we propose using a combination of these two approaches with what we call *pseudo-labeled data*. Instead of asking the builder to hand-label numerous training examples, or to generate a complete-coverage set of keywords, the builder simply provides just a few keywords for each category. A large collection of unlabeled documents are "pseudo-labeled" by using the keywords as a rule-list classifier. These pseudo-labels are noisy, and the majority of documents remain unlabeled. However, we then build an improved classifier by using all the documents and any pseudo-labels to bootstrap a naive Bayes text classifier that has been combined with Expectation-Maximization (EM) (Dempster, Laird, & Rubin 1977) and a powerful technique from statistics called *shrinkage*. EM serves to incorporate the evidence from the unlabeled data, and to correct, to some extent, the pseudo-labels. Hierarchical shrinkage serves to alleviate poor parameter estimates caused by sparse training data.

In this approach, using an enhanced naive Bayes text classifier acts to smooth the brittleness of the original keywords. One way to understand this is that naive Bayes discovers new keywords that are probabilistically correlated with the original keywords. The resulting pseudo-labeled method provides classification accuracy that is higher than keyword matching.

### 5.2 Naive Bayes Text Classification

We use the framework of multinomial naive Bayes text classification. The classifier parameterizes each class separately with a document frequency, and also word frequencies. 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 $c_j$.

We represent a document, $d_i$, as an unordered collection of its words. Given a document and a classifier, we determine the probability that the document belongs in class $c_j$ by 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:

$$\begin{aligned} \mathrm{P}(c_j|d_i) &\propto \mathrm{P}(c_j)\mathrm{P}(d_i|c_j) \\ &\propto \mathrm{P}(c_j)\prod_{k=1}^{|d_i|}\mathrm{P}(w_{d_{i,k}}|c_j). \end{aligned} \quad (5)$$

Learning these parameters ($\mathrm{P}(c_j)$ and $\mathrm{P}(w_t|c_j)$) for classification 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 that word $w_t$ occurs in 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. Define $N(w_t, d_i)$ to 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:

$$\mathrm{P}(w_t|c_j) = \frac{1 + \sum_{d_i \in \mathcal{D}} N(w_t, d_i)\mathrm{P}(c_j|d_i)}{|V| + \sum_{s=1}^{|V|}\sum_{d_i \in \mathcal{D}} N(w_s, d_i)\mathrm{P}(c_j|d_i)}. \quad (6)$$

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

$$\mathrm{P}(c_j) = \frac{1 + \sum_{d_i \in \mathcal{D}} \mathrm{P}(c_j|d_i)}{|\mathcal{C}| + |\mathcal{D}|}. \quad (7)$$

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 & Nigam (1998).

### 5.3 Combining Labeled and Unlabeled Data

When there are both labeled and unlabeled documents available when training a naive Bayes classi-

fier, Expectation-Maximization can be used to probabilistically fill in the missing class labels of the unlabeled data, allowing them to be used for training. This results in parameters that are more likely given all the data, both the labeled and the unlabeled. In previous work (Nigam *et al.* 1999), we have shown this technique significantly increases classification accuracy with limited amounts of labeled data and large amounts of unlabeled data.

EM is a class of iterative algorithms for maximum likelihood 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 unlabeled data are treated as the missing values.

In implementation, EM is an iterative two-step process. Initially, the parameter estimates are set in the standard naive Bayes way, using just the labeled data. Then we iterate the E- and M-steps. The E-step calculates probabilistically-weighted class labels, $\mathrm{P}(c_j|d_i)$, for every unlabeled document using the classifier and Equation 5. The M-step estimates new classifier parameters using all the labeled data, both original and probabilistically labeled, by Equations 6 and 7. We iterate the E- and M-steps until the classifier converges.

### 5.4 Shrinkage and Naive Bayes

When the classes are organized hierarchically, as they are in Cora, naive Bayes parameter estimates can also be significantly improved with the statistical technique *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 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 that are guaranteed to maximize the likelihood of held-out data by an iterative procedure.

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 estimate using training data just in the leaf, $\mathrm{P}^{k-1}(w_t|c_j)$ 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 $\sum_{i=1}^{k} \lambda_j^i = 1$. The new word probability estimate based on shrinkage, denoted $\check{\mathrm{P}}(w_t|c_j)$, is then

$$\check{\mathrm{P}}(w_t|c_j) = \lambda_j^1 \mathrm{P}^1(w_t|c_j) + \ldots + \lambda_j^k \mathrm{P}^k(w_t|c_j). \quad (8)$$

The $\lambda_j$ vectors are calculated using EM. In the E-step, for every word of training data in class $c_j$, we determine the expectation that each ancestor was responsible for generating it (in leave-one-out fashion, withholding from parameter estimation the data in the word's document). In the M-step, we normalize the sum of these expectations to obtain new mixture weights $\lambda_j$. Convergence usually occurs in less than 10 iterations and less than 5 minutes of wall clock time. A more complete description of hierarchical shrinkage for text classification is presented by McCallum *et al.* (1998).

### 5.5 Experimental Results

Now we describe results of classifying computer science research papers into our 70-leaf hierarchy. A test set was created by one expert hand-labeling 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 did not fit into any category, and were discarded—resulting in a 400 document test set. Some of these 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. In these experiments, we used the title, author, institution, references, and abstracts of papers for classification, not the full text.

Table 2 shows classification results with different techniques used. The rule-list classifier based on the keywords alone provides 45%. Traditional naive Bayes with 399 labeled training documents, tested in a leave-one-out fashion, results in 47% classification accuracy. However, only 100 documents could have been hand-labeled in the time it took to create the keyword-lists; using this smaller training set results in 30% accuracy. We now turn to our pseudo-label approach. Applying the keyword rule-list to the 30,682 documents in the

| Method | # Lab | # P-Lab | # Unlab | Acc |
|---|---|---|---|---|
| Keyword | — | — | — | 45% |
| NB | 100 | — | — | 30% |
| NB | 399 | — | — | 47% |
| NB | — | 12,657 | — | 47% |
| NB+S | — | 12,657 | — | 63% |
| NB+EM+S | — | 12,657 | 18,025 | 66% |
| Human | — | — | — | 72% |

Table 2: Classification results with different techniques: keyword matching, human agreement, naive Bayes (NB), and naive Bayes combined with hierarchical shrinkage (S), and EM. The classification accuracy (Acc), and the number of labeled (Lab), keyword-matched pseudo-labeled (P-Lab), and unlabeled (Unlab) documents used by each method are shown.

archive results in 12,657 matches, and thus an equivalent number of pseudo-labeled documents. When these noisy labels are used to train a traditional naive Bayes text classifier, 47% accuracy is reached on the test set. When naive Bayes is augmented with hierarchical shrinkage, accuracy increases to 63%. The full algorithm, including naive Bayes, shrinkage, and EM reassignment of the pseudo-labeled and unlabeled data, achieves 66% accuracy. As an interesting comparison, a second expert classified the same test set; human agreement between the two was 72%.

These results demonstrate the utility of the pseudo-label approach. Keyword matching alone is noisy, but when naive Bayes, shrinkage and EM are used together as a regularizer, the resulting classification accuracy is close to human agreement levels. We expect that using EM with unlabeled data will yield much larger benefits when the hierarchy is expanded to cover more of computer science. Approximately one-third of the unlabeled data do not fit the hierarchy; this mismatch of the data and the model misleads EM. The paradigm of creating pseudo-labels, either from keywords or other sources, avoids the significant human effort of hand-labeling training data.

In future work 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.

## 6 Related Work

Several related research projects investigate the gathering and organization of specialized information. The WebKB (Craven *et al.* 1998) project 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 (Bollacker, Lawrence, & Giles 1998) has also developed a search engine for computer science research papers. It provides similar functionality—matching references and allowing searches—but does not currently provide a hierarchy of the field. Previously published CiteSeer work concentrates on citation indexing and uses less machine learning than our approach; in ongoing work, however, CiteSeer's functionality and methods are expanding [Lawrence, Personal Communication].

The New Zealand Digital Library project (Witten *et al.* 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 search engines. We have presented new research in reinforcement learning, text classification and information extraction towards this

end.

Much future work in each machine learning area has already been discussed. However, we also see many other areas where machine learning can further automate the construction and maintenance of domain-specific search engines. 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. Citation graph analysis can identify seminal papers. We anticipate developing a suite of many machine learning techniques so domain-specific search engine creation can be accomplished quickly and easily.

## Acknowledgements

Most of the work in this paper was performed while all the authors were at Just Research. The second, third and fourth authors are listed in alphabetic order. Kamal Nigam was supported in part by the Darpa HPKB program under contract F30602-97-1-0215.

## References

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, NJ: Princeton University Press.

Bikel, D. M.; Miller, S.; Schwartz, R.; and Weischedel, R. 1997. Nymble: a high-performance learning name-finder. In *Proceedings of ANLP-97*, 194–201.

Bollacker, K. D.; Lawrence, S.; and Giles, C. L. 1998. CiteSeer: An autonomous web agent for automatic retrieval and identification of interesting publications. In *Agents '98*, 116–123.

Boyan, J.; Freitag, D.; and Joachims, T. 1996. A machine learning architecture for optimizing web search engines. In *AAAI workshop on Internet-Based Information Systems*.

Charniak, E. 1993. *Statistical Language Learning*. Cambridge, Massachusetts: The MIT Press.

Cho, J.; Garcia-Molina, H.; and Page, L. 1998. Efficient crawling through URL ordering. In *WWW7*.

Cohen, W., and 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 *Agents '98*.

Craven, M.; DiPasquo, D.; Freitag, D.; McCallum, A.; Mitchell, T.; Nigam, K.; and Slattery, S. 1998. Learning to extract symbolic knowledge from the World Wide Web. In *AAAI-98*, 509–516.

Dempster, A. P.; Laird, N. M.; and 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.

Hofmann, T., and Puzicha, J. 1998. Statistical models for co-occurrence data. Technical Report AI Memo 1625, Artificial Intelligence Laboratory, MIT.

Joachims, T.; Freitag, D.; and Mitchell, T. 1997. Webwatcher: A tour guide for the World Wide Web. In *Proceedings of IJCAI-97*.

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

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 *ECML-98*.

McCallum, A., and 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.; and Ng, A. 1998. Improving text clasification by shrinkage in a hierarchy of classes. In *ICML-98*, 359–367.

Menczer, F. 1997. ARACHNID: Adaptive retrieval agents choosing heuristic neighborhoods for information discovery. In *ICML '97*.

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

Nigam, K.; McCallum, A.; Thrun, S.; and Mitchell, T. 1999. Text classification from labeled and unlabeled documents using EM. *Machine Learning*. To appear.

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

Stolcke, A. 1994. *Bayesian Learning of Probabilistic Language Models*. Ph.D. Dissertation, UC Berkeley.

Torgo, L., and Gama, J. 1997. Regression using classification algorithms. *Intelligent Data Analysis* 1(4).

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.; and 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.; and van Mulbregt, P. 1998. A hidden Markov model approach to text segmentation and event tracking. In *Proceedings of the IEEE ICASSP*.