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 becoming increasingly popular because they offer increased accuracy and extra features 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, text classification and information extraction that automates efficient spidering, populating topic hierarchies, and identifying informative text segments. Using these techniques, we have built a demonstration system: a search engine for computer science research papers. It already contains over 33,000 papers and is publicly available at *www.cora.jprc.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:8000/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 the automation of three different aspects of search engine creation, using reinforcement learning, text classification and information extraction.

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 preliminary experimental results show that a simple reinforcement learning spider is nearly three times more efficient than a spider with a breadth-first search strategy.

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 the results of keyword matching to obtain training data with approximate class labels, and (2) performing robust parameter estimation in the face of sparse data by using a statistical technique called *shrinkage* that takes advantage of the hierarchy. Use of the resulting algorithms places documents into a 51-leaf computer science hierarchy with 70% accuracy—a figure we believe is not far below human disagreement. We also present preliminary results indicating that we can increase accuracy even further by augmenting the labeled training data with a large pool of unlabeled data, and integrating the two using Expectation-Maximization (Dempster, Laird, & Rubin 1977).

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 techniques from statistical language modeling and speech recognition, namely *hidden Markov models* (Rabiner 1989). Our initial algorithm extracts the title, authors, institution, journal name, etc., from research paper reference sections with 93% accuracy.

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.jprc.com*. Not only does it provide phrase and keyword search facilities over 33,000 collected papers, it also places these papers into a computer science topic hierarchy, maps the web of citations 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—complementing 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. 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 the repository. The ASCII text is extracted from the postscript document. Using this system, we have found 33,000 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, date, etc. Using this extracted information, reference and paper matches are made—grouping all citations to the same paper together, and matching citations to papers from the repository. Of course, many papers that are cited do not appear in the repository. This matching procedure is similar to one used by CiteSeer (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

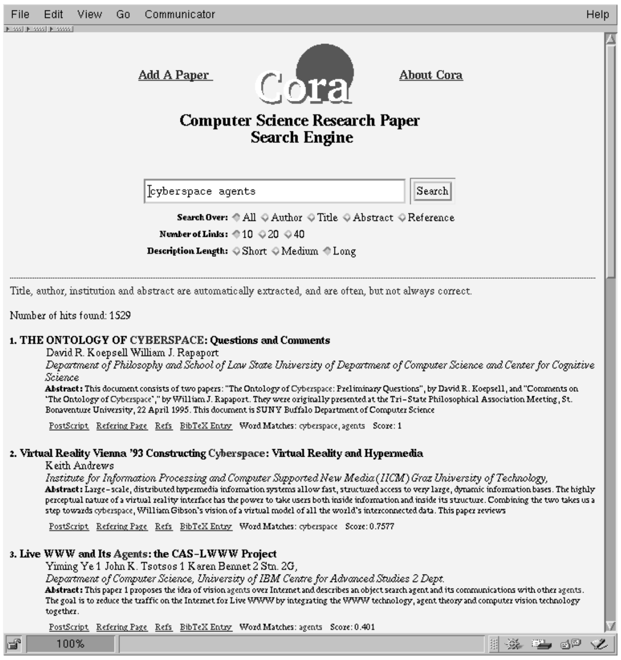

Figure 1: A screen shot of the query results page of the Cora search engine (*www.cora.jprc.com*). Note that paper titles, authors and abstracts are provided at this level.

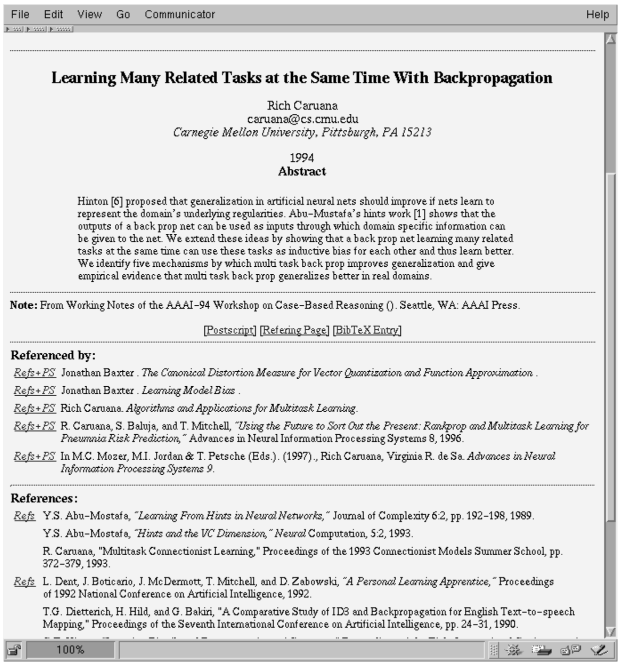

Figure 2: A screen shot of a details page of the Cora search engine. At this level, all extracted information about a paper is displayed, including the citation linking, which are hyperlinks to other details pages.

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 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 51 leaves, varying in depth from one to three. Each leaf node in the hierarchy is seeded with just a few keywords, and from these keywords a robust, hierarchical naive Bayes classifier is built. Using the classifier, 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 efficient information gathering from the Web. ARACHNID (Menczer 1997) maintains a collection of competitive, reproducing and mutating agents for finding information on the Web. WebWatcher (Joachims, Freitag, & Mitchell 1997) is a browsing assistant that helps a user find information by recommending which hyperlinks to take next using reinforcement learning. 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. Laser uses reinforcement learning to tune the search 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 π, 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.[1] 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:

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

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 straightforwardly found by dynamic programming.

[1] In preliminary experiments currently reported we restrict $\gamma = 0$; experiments in progress are using delayed rewards.

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 locations of the on-topic documents remaining 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; more specifically, we begin by assuming that the value of an action is a function of the unordered list of words in the anchor text of the hyperlink.

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 straightforwardly 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 4.2). 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 ten bins, place the hyperlinks into the bin corresponding to their Q values as calculated above, and use the hyperlinks' text as training data for a naive Bayes text classifier. For the anchor text of each hyperlink, we calculate the probabilistic class membership for each bin. Then the reward value of a hyperlink is set 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.[2]

The value of all the hyperlinks is determined by finding all target pages, and propagating reward out along the incoming hyperlinks, with dynamic programming. Experiments with delayed reward are in progress. We currently report results with immediate reward only, where $\gamma = 0$. This assignment results in the degenerate case of dynamic programming where in the first and only iteration the value of a hyperlink is set to the number of target pages pointed to by the destination of the hyperlink.

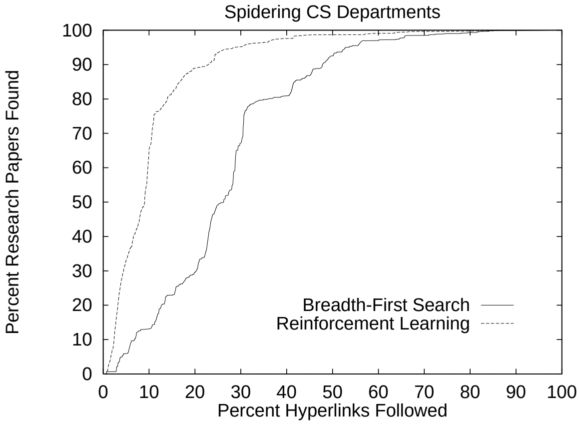

Figure 3: The performance of reinforcement learning spidering versus traditional breadth-first search, averaged over four test/train splits with data from four universities. The vertical axis shows the percentage of on-topic documents found, while the horizontal axis shows the percentage of hyperlinks followed thus far into the spider's exploration. The reinforcement learning spider finds target documents significantly faster than the traditional method.

A spider trained in this fashion is evaluated on each test/train split by having it spider the test university, and performance is compared with breadth-first search. Figure 3 plots the number of research papers found over the course of all the pages visited, again averaged over all four universities. Notice that at all times during its progress, the reinforcement learning spider has found more research papers than breadth-first search, and that its performance is especially strong in the beginning.

One measure of performance is the number of hyperlinks followed before 75% of the research papers are found. Reinforcement learning performance is significantly more efficient, requiring exploration of only 11% of the hyperlinks, in comparison to the breadth-first search's 30%. This represents nearly a factor of three increase in spidering efficiency.

3.5 Future Work

The most important next step after these preliminary results is to relax some of the restrictive assumptions made thus far. Towards this goal, experiments aim to show that representing delayed reward can further improve the efficiency of a directed spider. We believe that there are many features that are not indicative of immediate reward but are predictive of future reward; following them should improve performance further. We are also studying enlarged feature sets to represent a hyperlink for the purpose of the Q function: namely, we are working on using word neighborhoods,

[2] They are identified by a separate, simple hand-coded algorithm that has very high precision.

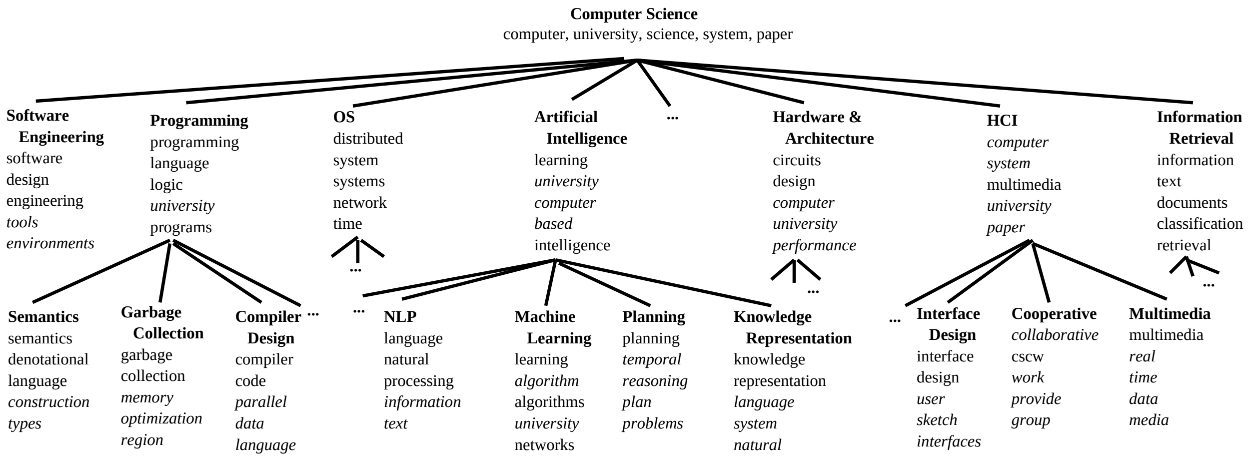

Figure 4: A subset of Cora's topic hierarchy. Each node contains its title, and the five most probable words, as calculated by naive Bayes and special shrinkage with vertical word redistribution. Several additional nodes in deeper layers of the hierarchy are not shown due to space limitations. Words that were not among the keywords for that class are indicated with italics.

headers, titles, and words from hyperlink-neighboring pages. Finally, we are also exploring methods that include network latency in the optimization criteria, so that fast web servers can be explored before slower ones.

4 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 are created 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 51-leaf hierarchy of computer science topics for Cora, shown in part in Figure 3.5. Creating the hierarchy and selecting just a few keywords associated with each node took about three hours, during which we examined conference proceedings, and explored computer science sites on the Web.

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. People make careers of categorizing publications into the Dewey Decimal system. In contrast, we automate this process with learned text classifiers.

4.1 Seeding Naive Bayes using Keywords

One method of classifying documents into a hierarchy is to match them against the keywords: for each document, step through the keywords, and place the document in the category of the first keyword that matches. If the keywords are carefully chosen, this method can be surprisingly accurate. However, finding enough keywords to obtain broad coverage and finding sufficiently specific keywords to obtain high accuracy can be 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 the specific list of words to match.

A less brittle approach is provided by naive Bayes, an established text classification algorithm (Lewis 1998; McCallum & Nigam 1998) based on Bayesian techniques of machine learning. 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.

We propose to combine these two approaches by using keyword-matching as a method of inexpensively obtaining imperfectly-labeled documents, and then using these documents as training data for a naive Bayes classifier. In this case, naive Bayes 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.

Unfortunately, even with the training data provided by keyword-matching, naive Bayes can still suffer from sparseness in the training data. We overcome this problem by combining naive Bayes with a powerful technique from statistics called *shrinkage*. The resulting method provides classification accuracy that is higher than keyword matching. We also discuss preliminary results indicating strong promise for further improvement by adding a third technique: Expectation-Maximization with unlabeled data.

4.2 Naive Bayes Text Classification

Naive Bayes approaches the task of text classification from a Bayesian learning framework. It assumes that text data is generated by a parametric model, and uses training data to calculate estimates of the model parameters. Equipped with these estimates, it classifies new test documents using Bayes' rule to turn the generative model around and calculate the posterior probability that each class would have generated the test document in question.

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 $\mathrm{P}(c_j)$. Each class is modeled by a multinomial over words. That is, 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. To classify a new document with this model, we make the naive Bayes assumption: that the words in the document occur independently of each other given the class of the document, (and furthermore independently of position). Using this assumption, classification becomes straightforward. We calculate the probability of each class, given the evidence of the document, $\mathrm{P}(c_j|d_i)$, and select the class for which this expression is the maximum. We denote $w_{d_{ik}}$ to be the kth word in document d. We expand $\mathrm{P}(c_j|d_i)$ with an application of Bayes' rule, and then make use of the word independence assumption:

$$\begin{aligned} \mathrm{P}(c_j|d_i) &\propto \mathrm{P}(c_j)\mathrm{P}(d_i|c_j) \\ &\propto \mathrm{P}(c_j)\prod_{k=1}^{|d_i|}\mathrm{P}(w_{d_{ik}}|c_j). \end{aligned} \quad (3)$$

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 over all word occurrences for class c_j the frequency that w_t occurs in documents from that class. 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:

$$\mathrm{P}(w_t|c_j) = \frac{1 + \sum_{d_i\in\mathcal{D}} N(w_t, d_i)\mathrm{P}(c_j|d_i)}{|V| + \sum_{s=1}^{|V|}\sum_{d_i\in\mathcal{D}} N(w_s, d_i)\mathrm{P}(c_j|d_i)}. \quad (4)$$

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

$$\mathrm{P}(c_j) = \frac{1 + \sum_{d_i\in\mathcal{D}} \mathrm{P}(c_j|d_i)}{|\mathcal{C}| + |\mathcal{D}|}. \quad (5)$$

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 Nigam *et al.* (1999).

4.3 Shrinkage and Naive Bayes

When naive Bayes is not provided with sufficient training data, the parameters for word probability estimates will be poor. We now describe shrinkage—a method for improving these estimates by taking advantage of the hierarchy. In our hierarchical setting, consider trying to estimate the probability of the word "intelligence" in the class NLP. Clearly this is a word that should have non-negligible probability in NLP. However, when the amount of training data is small, we may be unlucky and the observed frequency of "intelligence" may be very far from its true value. However, one level up in the hierarchy, the Artificial Intelligence class contains all the NLP documents, plus many more somewhat related ones. Here, the probability of the word "intelligence" may be reliably estimated. This estimate, while related, is not the true estimate for the NLP class. Shrinkage uses a *weighted average* of estimates along an entire path from a leaf to the root, where leaf-level estimates are the most specific, but unreliable, and the highest-level estimates are the most reliable, but unspecific. We can calculate mixture weights guaranteed to maximize the likelihood of held-out data by an iterative re-calculation process for the weights.

More formally, let $\{\mathrm{P}^1(w_t|c_j), \ldots, \mathrm{P}^k(w_t|c_j)\}$ be k such 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 the ancestors of class c_j are written $\{\lambda_j^1, \lambda_j^2, \ldots, \lambda_j^k\}$, where $\sum_{i=1}^k \lambda_j^i = 1$.

We write $\check{\mathrm{P}}(w_t|c_j)$ for the new estimate of the class-conditioned word probabilities based on shrinkage. The new estimate for the probability of word w_t given class c_j is just the weighted average of the estimates along the path to the root of the hierarchy:

$$\check{\mathrm{P}}(w_t|c_j) = \lambda_j^1 \mathrm{P}^1(w_t|c_j) + \ldots + \lambda_j^k \mathrm{P}^k(w_t|c_j). \quad (6)$$

Given a set of parameter estimates along the path from a leaf to the root, how do we decide on the weights for performing the weighted average? The mixture weights, λ_j^i are set such that they maximize the likelihood of some previously unseen "held-out" data by using EM (Dempster, Laird, & Rubin 1977). This method straightforwardly calculates such weights for each leaf class, c_j, using the following iterative procedure:

First, for each class c_j, sum, (over each word in the held-out data), the likelihood that it was generated by the ith ancestor. Call this sum β_j^i:

$$\beta_j^i = \sum_{w_t \in d_i \in \mathcal{D}} \lambda_j^i \mathrm{P}^i(w_t|c_j)\mathrm{P}(c_j|d_i). \quad (7)$$

Then, normalize all the β_j's (on a path from a leaf node to the root) such that they sum to one. Then, set the new value of each λ_j^i to its corresponding normalized β_j^i, and iterate until the λ's converge to a stable value, usually in less than a dozen iterations.

A more complete description of hierarchical shrinkage for text classification is presented by McCallum *et al.* (1998). Carlin & Louis (1996) present a introduction and summary of general shrinkage.

The experimental results reported in the next section use a special shrinkage model with an extra degree of freedom: the EM procedure is used not only to set the mixture weights λ, but also to redistribute the word data up and down the hierarchy. Space limitations prevent a full explanation here, but details are given by Hofmann & Puzicha (1998).

4.4 Experimental Results

Now we describe results of classifying computer science research papers into the 51-leaf hierarchy mentioned above. A test set was created by taking a random sample of 250 research papers from the 33,000 papers currently in the Cora archive. These 250 papers were then categorized into the leaves of the topic hierarchy by hand. Forty of them did not fit into any of the leaves, and were discarded—resulting in a 205 document test set. Keyword matching was applied to the documents remaining in the archive, and, by taking the top 40 matches per leaf (as measured by our information retrieval engine) 4,631 documents with matches were found. In these experiments, we used only the title, author, institution, and abstracts of papers, and not the full text of the paper.

The keyword-matching method obtained a surprisingly good 66% accuracy. If we were not working in a domain with which we already have much expertise, we expect that good keywords would be more elusive, and that reduced accuracy would result. Traditional naive Bayes, when trained on these imperfectly-labeled documents receives a reduced accuracy of 62%. Here, the median number of these short training documents is only 100 per class, which is quite small for estimating the parameters of a multinomial with a vocabulary size of 58079. The fact that data sparseness limits its accuracy here is confirmed by the increased performance of shrinkage. It achieves 70% accuracy, providing our highest accuracy yet. A beneficial feature of shrinkage is that as the hierarchy becomes wider and deeper, shrinkage should improve relative performance even more strongly because fragmentation will cause per-class data to be even more sparse, and because there will be more nodes in the hierarchy from which to "borrow strength."

Another interesting source of extra training data is completely unlabeled data—documents that were not matched by any of the keywords. In past work (Nigam *et al.* 1999) we show that text classification accuracy can be dramatically increased by augmenting a small set of labeled data with a large pool of unlabeled data. EM is used to probabilistically fill in the "missing" class labels. We have recently performed a preliminary experiment showing that this technique holds promise for the Cora classification task. We randomly selected 464 out of the 4631 keyword-labeled documents. When these are used to train traditional naive Bayes, it obtains 29% accuracy. The application of shrinkage increases accuracy to 47%. Then, interestingly, incorporating the remaining 4167 documents as unlabeled data brings accuracy up to an impressive 56%. In upcoming work we will perform experiments applying this technique to large quantities of the 33,000 Cora documents.

5 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 extraction of information specific to the domain of interest can increase the accuracy and efficiency of a directed search.

We have investigated techniques for extracting the fields relevant to research papers, such as title, author, journal and publication date. The extracted fields are used to allow searches over specific fields, to provide useful effective presentation of search results (*e.g.*

showing title in bold), and to match references to papers, in order to display links to papers referenced by the current paper, and papers that reference the current paper.

5.1 Hidden Markov Models

Our information extraction approach is based on hidden Markov models (HMMs) and their accompanying search techniques that are so widely used for speech recognition and part-of-speech tagging (Rabiner 1989; Charniak 1993). Discrete output, first-order hidden Markov models 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 alphabet of output symbols $\Sigma = \sigma_1 \sigma_2 \ldots \sigma_M$. The model generates strings $x = x_1 x_2 \ldots x_l$ by beginning in the initial state, transitioning to 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 x being emitted by an HMM M is computed by:

$$P(x|M) = \sum_{q_1,\ldots,q_l \in Q^l} \prod_{k=1}^{l+1} \mathrm{P}(q_{k-1} \rightarrow q_k)\mathrm{P}(q_k \uparrow x_k), \quad (8)$$

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 $V(x|M)$ that has the highest probability of having produced an observation sequence:

$$V(x|M) = \mathop{\arg\max}_{q_1 \ldots q_l \in Q^l} \prod_{k=1}^{l+1} \mathrm{P}(q_{k-1} \rightarrow q_k)\mathrm{P}(q_k \uparrow x_k). \quad (9)$$

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, institution, etc. Each state emits words from a class-specific unigram distribution. In order to label classes from new text, we treat the words from the new text as observations and use the Viterbi algorithm to recover the most-likely state sequence. Any words in the Viterbi path produced by the "title" state are labeled as title words, and so on. We can learn the class-specific unigram distributions and the transition probabilities from data. In our case, we collected BibTeX files from the Web with reference classes explicitly labeled, and used the text from each class as training data for the appropriate unigram model. Transitions between states are estimated directly from a labeled training set.

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. Our approach to information extraction is similar to wrapper induction (Knoblock *et al.* 1998; Kushmerick 1997).

5.2 Experiments

In order to investigate the modeling potential of HMMs for information extraction from research papers, we conducted the following set of experiments on reference extraction. Five hundred references 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. The unigram models were based on a vocabulary of 44,000 words, and were smoothed using absolute discounting (Ney, Essen, & Kneser 1994). For all of the models we considered, every state was always associated with one class label, and used the appropriate unigram distribution to provide its emission probabilities. Emission distributions were never re-estimated during the training process.

Four increasingly sophisticated modeling techniques were investigated, all using the labels provided in the training data. Techniques to derive HMMs that do not rely on labeled training data are discussed below, and results using these techniques are forthcoming.

The first experiment was to create a fully-connected HMM (HMM-0) where each class was represented by a single state. The outgoing transitions for each state were given equal probability. Finding the most likely

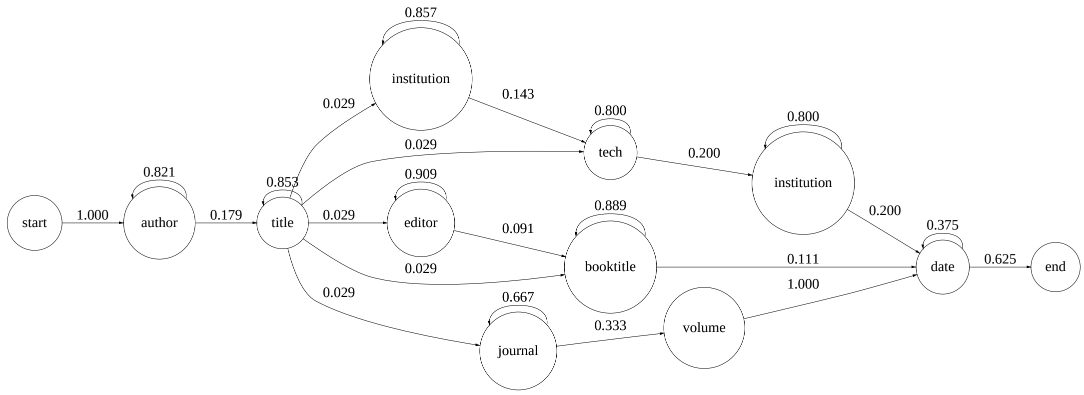

Figure 5: HMM built from only five labeled references after merging neighbors and collapsing V-neighbors in the forward and backward directions. Note that the structure is close to many reference section formats.

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.

Second, another fully connected HMM (HMM-1) was built with one state per class, except that the transition probabilities were estimated from the labeled training data. In this case, the transition parameters were set to the maximum likelihood estimates determined by counting the transitions in the 300 tagged training references, plus a smoothing count of 1 which was added to all transitions to avoid non-zero probabilities.

Next, an HMM was built directly from the class labels of the tagged training references. Each word token in the training set, represented by its class label, was assigned a single state that only transitioned to the state that followed it. From the initial state, there were 300 equiprobable transitions into sequences of states, where each sequence represented the tags for one of the 300 training references. This model consisted of 6997 states, and was maximally specific in that its transitions exactly explained the training data.

This maximally specific HMM was put through a series of state merges in order to generalize the model. First, "neighbor merging" combined all states that shared a unique transition and had the same class label. For example, all adjacent title states for one reference are merged into one title state, representing the sequence of title words for that reference. As two states were merged, transition counts were preserved, so that a self-loop was introduced on the new merged state. As multiple neighbor states with the same class label were merged into one, the self-transition loop probability increased, and represented the expected state duration for that class. After neighbor merging, the maximally specific HMM was reduced from 6997 states to 1677 states (HMM-2).

Next, the neighbor-merged HMM was 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. An example of an HMM built from just 5 tagged references after V-merging is shown in Figure 5[3]. Notice that even with just five references, the model closely matches the formats found in many reference sections. After V-merging, the HMM was reduced from 1677 states to 46 states (HMM-3).

All four HMM models were 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 were allowed to occur after any observation word. In the 'Punc word' case, state transitions to a new state (with a different class label) were only allowed to occur after observations ending in punctuation. In many format styles, punctuation is a standard delimiter for fields. 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 deterministic HMM where only one state was allowed per class (HMM-1). For these three cases, limiting state transitions to occur only after words with

[3]The HMM Graph was produced with AT&T's graphviz package, available at http://www.research.att.com/sw/tools/graphviz/

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).

punctuation improved accuracy by about 1% absolute.

5.3 Future Work

All of the experiments presented above used HMMs where the model structure and parameters were estimated directly from labeled training instances. In the near future, we will begin using model estimation techniques that do not rely on labeled training examples to induce a model. Using unlabeled 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 training data. Additionally, manually labeling large amounts of training data can be costly and error-prone.

Specifically, if we are willing to fix the model size and 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 also remove the assumption of a fixed model size and estimate the model size, 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 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.

6 Related Work

Several related research projects are investigating the automatic construction of special-purpose web sites. Perhaps the most related is the New Zealand Digital Library project (Witten *et al.* 1998), which 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 underlying machine learning technology. 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 WebKB project (Craven *et al.* 1998) is an effort to extract domain-specific information available on the Web into a knowledge base. 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. No searching facilities are provided over the extracted knowledge bases.

The CiteSeer project (Bollacker, Lawrence, & Giles 1998) has also developed an internal search engine for computer science research papers. It provides similar functionality for searching and linking of research papers. It does not provide information extraction of papers and references, or a hierarchy of the field. The CiteSeer project is aimed at reproducing a citation index, and thus focuses more on domain-specific implementation aspects for research papers, and not as much on automating the general construction of search engines with machine learning techniques.

The WHIRL project (Cohen 1998) is an effort to integrate a variety of topic-specific sources into a single domain-specific search engine. Information is extracted from web pages by simple hand-written extraction patterns that are customized for each web source. The emphasis is on providing fuzzy matching for information retrieval searching. Two demonstration domains of computer games and North American birds integrate information from tens of web sites each.

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 though this information, but the *creators* of these tools will need intelligent techniques to help them build and maintain the tools. This paper has shown that machine learning techniques can significantly aid the creation and maintenance of domain-specific search engines. We have presented 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. Collaborative filtering and information retrieval can generate a user-specific recommended reading list. 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.

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*.

Carlin, B., and Louis, T. 1996. *Bayes and Empirical Bayes Methods for Data Analysis*. Chapman and Hall.

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. 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.

Knoblock, C.; Minton, S.; Ambite, J. L.; Ashish, N.; Modi, P.; Muslea, I.; Philpot, A. G.; and Tejada, S. 1998. Modeling web sources for information integration. In *AAAI-98*.

Kushmerick, N. 1997. *Wrapper Induction for Information Extraction*. Ph.D. Dissertation, University of Washington.

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.

Ney, H.; Essen, U.; and Kneser, R. 1994. On structuring probabilistic dependences in stochastic language modeling. *Computer, Speech and Language* 8:1–38.

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*.