Creating a Web Page Recommendation System For Haystack

by

Jonathan C. Derryberry

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2003

Author . . . .
Department of Electrical Engineering and Computer Science
September 17, 2003

Certified by . . . .
David Karger
Associate Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Creating a Web Page Recommendation System For Haystack

by

Jonathan C. Derryberry

Submitted to the Department of Electrical Engineering and Computer Science
on September 17, 2003, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The driving goal of this thesis was to create a web page recommendation system for Haystack, capable of tracking a user's browsing behavior and suggesting new, interesting web pages to read based on the past behavior. However, during the course of this thesis, 3 salient subgoals were met. First, Haystack's learning framework was unified so that, for example, different types of binary classifiers could be used with black box access under a single interface, regardless of whether they were text learning algorithms or image classifiers. Second, a tree learning module, capable of using hierarchical descriptions of objects and their labels to classify new objects, was designed and implemented. Third, Haystack's learning framework and existing user history faculties were leveraged to create a web page recommendation system that uses the history of a user's visits to web pages to produce recommendations of unvisited links from user-specified web pages. Testing of the recommendation system suggests that using tree learners with both the URL and tabular location of a web page's link as taxonomic descriptions yields a recommender that significantly outperforms traditional, text-based systems.

Thesis Supervisor: David Karger
Title: Associate Professor

Acknowledgments

First, I would like to thank my supervisor Professor David Karger for suggesting this topic and for providing me guidance along the way. Also, Kai Shih helped me during this thesis by discussing his work with tree learning algorithms and providing me with test data from a user study he administered. Additionally, the Haystack research group was invaluable in helping me familiarize myself with the Haystack system. David Huynh and Vineet Sinha were of particular aid in teaching me how to accomplish various coding tasks in the Haystack system and in helping me debug the Haystack-dependent code that I wrote. Finally, thanks to Kaitlin Kalna for being patient and providing welcome diversions from my thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter introduces the context of this thesis, providing basic information about the Haystack information management project and the subfield of web page recommendation, which is the driving application of this thesis.

1.1 Haystack

The Haystack project was started because of the increasing need to help users interact with their ever growing corpora of data objects, such as e-mails, papers, calendars, and media files. As opposed to traditional user interfaces, in which distinct applications are used to interact with different types of files using different user interfaces that are each constrained in the way the users view and manipulate their data, Haystack attempts to unify all interaction into a single application. Haystack allows users to organize their data according to its semantic content, rather than its superficial file type.

This unification is achieved by creating a *semantic user interface* whereby each data object can be annotated with arbitrary metadata in subject-predicate-object format. For example, the paper “Factoring in Polynomial Time” could be listed as having author Ben Bitdiddle, being a member of a set of papers to be presented in at a seminar in the near future, or being in a list of the user’s favorite files. The semantic user interface’s job is to render data objects given the context in which the user is browsing to them, and provide appropriate options in the menus for the objects.

In addition to providing a uniform, semantic user interface, Haystack provides an architecture for incorporating agents, which are autonomous programs that perform diverse activities such as gathering weather data, automatically classifying data objects into categories, and extracting text from documents. The agent architecture is an excellent setting for creating modules with a large amount of utility for the system. It allows the system to incorporate arbitrarily rich functionality by simply plugging in new agents.

1.2 Web Page Recommendation

In keeping with the underlying goal of helping the user manage large amounts of data, the goal of this thesis is to create an agent for Haystack that searches for new web pages that the user will likely find interesting. In doing so, the agent will decrease the effort a user has to expend navigating to interesting web documents, as well as help the user find both a greater quantity and/or quality of pages.

The recommendation of web pages is an area of active research. Most work in web page recommendation has used textual analysis (See Chapter 2). However, recently, Kai Shih and David Karger have proposed a new feature for learning about web pages [7]. They claim that the URL of web pages bears correlation to many interesting features of a page, such as whether the page is an advertisement and whether the user is interested in the page. They argue that the nature of this correlation is hierarchical; pages with similar URLs, such as the following two

- `www.cnn.com/2003/LAW/06/29/scotus.future`
- `www.cnn.com/2003/LAW/06/29/scotus.review`,

are likely to be equally interesting or uninteresting to the user.

Of course, correlation is not necessarily related to the URL's hierarchical description of the page. For example, consider the following two articles:

- `www.cnn.com/2003/WORLD/asiapcf/east/07/05/sars`

- `www.cnn.com/2003/HEALTH/conditions/07/05/sars`.

Both are articles about the SARS disease outbreak and may both be interesting to the user for that reason. Therefore, learning may be able to be improved by incorporating traditional textual methods.

In addition to the URL, the location of a link on a web page (e.g. first table, fourth element, second subtable, fifth element) is likely to be correlated to a user's interest in the link's contents. To make web pages more useful to visitors, web editors have an incentive to localize links to web pages that users will be interested in, because a user is unlikely to scrutinize an entire web page to examine all of its links. For example, visitors to CNN's home page may be too busy to look at all of its approximately 100 links; they may only look at headline news stories or at links that appear under their favorite subheading.

This thesis seeks to implement the web page recommending capability of Shih's "The Daily You" application by exploring the benefits of traditional textual analysis, as well as implementing the idea of learning over a hierarchy using the URL and tabular location of links. In addition, it seeks to provide robust interfaces for the recommendation of web pages that will allow new algorithms to be plugged into the recommender agent without requiring the whole agent to be rebuilt from scratch.

Chapter 2

Related Work

This chapter describes Kai Shih's The Daily You application, which was the motivation for this thesis, as well as a few other web page recommending systems that have been built.

2.1 The Daily You

The Daily You is an application that Kai Shih developed to recommend new interesting web pages [7]. The user gives the application a list of web pages to track, and uses the history of a user's clicks to provide a set of positive samples, which serves as the training data. Then, using both the URL and the tabular position of links, it recommends the top stories from each page it is tracking.

In addition, The Daily You cleans the pages that are displayed by blocking ads and static portions of web pages, such as the table of links to the various sections of a news website.

The algorithm that The Daily You uses to learn trends in URLs and tabular locations of links is a tree learning algorithm that uses a tree-shaped Bayesian network (See Chapter 3 for more details).

The high level functionality of the agent created in this thesis essentially emulates the web page recommending functionality of The Daily You, using a generalized version of Shih's algorithm as well as other tree learning algorithms developed in this thesis, although it is not necessary for the agent created in this thesis to exclusively use tree learning algorithms.

2.2 News Dude

Billsus and Pazzani's *News Dude* recommends news stories [2], with some emphasis placed on the development of a speech interface. To learn which stories are interesting, it uses the well-known Naïve Bayes and nearest neighbor algorithms on the text of the stories.

It is similar to the agent built in this thesis in that it customizes recommendations to the individual user. However, it is different in that it only uses textual analysis of the news stories and in that it *requires* active feedback. By contrast, the recommender agent in this thesis learns user preferences by using *any* binary classifier, including tree learners that do not even use text. Also, it uses passive feedback, while allowing active feedback via Haystack's categorization scheme, so that users are not required to exert any effort for the agent to learn.

One other interesting aspect of News Dude is that it attempts to differentiate short-term user interests from long-term interests, so that it captures fleeting trends in users' tastes. The agent built in this thesis does not attempt to make such a distinction.

2.3 MONTAGE

Corin Anderson and Eric Horvitz developed the MONTAGE system, which tracks users' routine browsing patterns and assembles a start page that contains links and content from other pages that the system predicts its users are likely to visit [1].

One of their important ideas for the learning process of MONTAGE was to note that the content that users find interesting varies depending on the context of the web access. In particular, their application uses not only the URLs of websites that are visited but also the time of day during which the access occurs. As a result, recommendations may change as the time of day changes.

Another interesting aspect of MONTAGE is that when agglomerating the start page, it considers not only the probability of a user's visiting the page in the current context, but also the navigation effort required to reach the page. Thus, between two pages that are equally likely to be visited, MONTAGE fetches the one that would require more browsing effort to reach.

Compared with the agent built in this thesis, MONTAGE is similar in that it uses features of web pages other than the text when recommending content to users. However, the underlying goal of the recommender agent is not exactly the same as MONTAGE. While MONTAGE strives to assist users in routing browsing to various web pages that are frequently visited, the recommender agent searches for links to new stories that have not yet been visited.

2.4 NewsWeeder

Ken Lang's NewsWeeder project is a netnews-filtering system that recommends documents leveraging both content-based and collaborative filtering, although the emphasis is on content [5]. NewsWeeder asks users for active feedback, which it uses in conjunction with textual analysis to improve future recommendations to its users. Lang introduces a Minimum Description Length heuristic for classifying new documents by their text, which he reports compares favorably to traditional textual analysis techniques.

NewsWeeder differs from the agent in this thesis in that it requires active feedback rather than simply allowing it, and in that uses only textual analysis to learn preferences.

Chapter 3

Background

This chapter introduces notation, terminology, and concepts that are essential to learn in order to fully understand the rest of this thesis.

3.1 Notation

Some custom notation is adopted to simplify probabilistic analysis of the tree learning algorithms in this thesis. This section introduces the notation for representing probabilities and other values associated with a tree learner, such as the discrete Bayesian tree learner described in Section 3.4.

Trees will be described as if their roots are at the top, so that children are further down in the tree and a node's parent is higher up. The variables i, j, and k will usually be used to represent classes to which a node may belong, while the variables x, y, and z will usually represent specific nodes in the tree. Expressions of the form x^j will be used to represent the event that the specified node (in this case x) is of the specified class (in this case j). The value p_x^i will be used to represent the probability of all of the evidence in the subtree rooted at node x, conditioned on the hypothesis x^i, that x is of class i. The value p^k will be used to represent the prior probability that the root of the tree belongs to class k.

3.2 Probability

This section provides an overview of some of the basic probability concepts that are necessary to understand the algorithms in this thesis. David Heckerman has written an excellent tutorial on learning with Bayesian networks, of which the original tree learning algorithm is a specific example [4]. This tutorial was consulted for ideas about how to improve tree learning.

First, it is essential to understand Bayes Rule in the following form

$$\Pr[A|B]\Pr[B] = \Pr[A \wedge B] = \Pr[B|A]\Pr[A]. \tag{3.1}$$

In particular, this rule implies that the probability of event A occurring given that B has occurred can be computed by the probability of both A and B occurring divided by the probability of B occurring. This implication is used by the original discrete Bayesian algorithm introduced in Section 3.4.

Second, it is important to have a general understanding of the Beta distribution, which is used, for example, in the discretized continuous Bayesian tree learner implemented in this thesis. In that tree learner, the Beta distribution is used as the probability distribution of each node's children (See Section 5.2). Although there is a lot that can be said about the derivation of the Beta distribution, its expectation, and how to set its prior, understanding these is not essential to understanding how it is used in this thesis. Therefore, only the basic characteristics of the Beta distribution are discussed here.

Basically, the Beta distribution is a distribution that is only positive for values in the interval $[0, 1]$. The shape of a Beta distribution is a quasi bell curve, centered around its expectation, which can be computed from the number of positive samples n_1 and negative samples n_0 upon which the distribution is based. The expected fraction of Beta distribution samples belonging to the positive class has the simple closed form

$$\frac{n_1}{n_1 + n_0}. \tag{3.2}$$

A Beta distribution has 2 important parameters for establishing a prior distribution:

- α_0, which represents the number of "virtual negative samples"
- α_1, which represents the number of "virtual positive samples"

The expected fraction of Beta distribution samples belonging to the positive class has the simple closed form

$$\frac{n_1 + \alpha_1}{n_1 + n_0 + \alpha_1 + \alpha_0} \tag{3.3}$$

when a prior is used.

The Beta distribution is commonly used in applications because it has positive values exactly on the $[0, 1]$ interval, its expectation has a simple formula, and because its prior can be set intuitively by adding "virtual samples."

3.3 Model for User Interaction

In this section, a model for user interaction is presented that motivates the development of tree learning and provides a concrete model against which to judge the aspects of various tree learning algorithms.

3.3.1 Labeling Links

Suppose a user browses a front page, such as CNN's home page every day. During that time, the user will click on some subset of the links that appear on the page that day. The goal of web page recommendation is to determine which links the user will click during the next day, when many new links are available. Although the user has not provided any feedback regarding new links, there may be feedback for many similar old links, such as links from the same part of the page or having similar URLs. To compute the probability that a new page is interesting, it would be intuitive to use the fraction of similar old links that were visited. For example, if there are 12 similar URLs and 3 were clicked, then it is reasonable to believe that there is about a 25% probability that the new link will be clicked.

One question that remains is how positive and negative examples should be added. One strategy is simply labeling nodes as clicked versus not clicked. In such a strategy, unvisited links would be used as negative samples until the user visits them, if ever, while repeated visits to the same page would be ignored. This strategy is simple and intuitive, but there are other aspects of web pages that merit discussion.

The simple setup ignores the fact that some nodes have more "facetime" than others. For example, a link to a section heading, such as `http://www.cnn.com/WORLD/`, may persist indefinitely, and if the user ever clicks on this link then it will be labeled as good, even if the one time that the user visited the link was by accident.

To provide a simple example of how this can cause problems, assume that a user clicks on all of these section headings at least once, even though none of them is particularly interesting. If a new section heading appears, it will be heavily supported because 100% of the similar links were clicked. On the other hand, if a user reads 60% of all of the health articles that are posted, then the user is likely more interested in a new health article than in a new section heading.

This issue motivates the practice of adding multiple pieces of evidence to the same link, one for each time the page is sampled. For example, suppose the page is sampled once day, and every link that has been clicked during the day has a positive label added to it and every page that has not been clicked has a negative label added to it. Then, 60% of the health labels will be positive, while many negative samples mount for the static, section links, so that perhaps only 5% of them may be positive.

The implementation of passive evidence gathering in this thesis is that each time the agent runs, all visited links are gathered as positive samples for the classifiers, and an equal number of negative samples are randomly picked off of the pages being tracked, from the subset of links that were not visited since the last time the agent ran. This choice retains all positive samples while keeping the total number of samples manageable because the number of samples is proportional to the number of times a human clicked on links.

As an aside, if the goal is to recommend pages that the user wants to read, then using clicks as a proxy is not a perfect solution because a user's clicks do not necessarily directly lead to interesting web pages. For instance, the user may mistakenly open a page, or use a link as an intermediate point while navigating to an interesting page. However, to avoid requiring active feedback, it was deemed the best proxy to use for passive feedback.

3.3.2 Hierarchical Correlation

New nodes are likely to have many nearby nodes in the tree with many pieces of evidence. However, when new nodes appear that do not have many immediate neighbors, the hierarchical correlation assumption can be used to to provide prior knowledge about the likelihood that the new node is in each class. The hierarchical correlation assumption states that children and parents are likely to be of the same class.

The reason why the assumption about hierarchical correlation is expected to hold when the tabular location of a link is used as a feature is that web editors have an incentive to localize stories of similar degrees of interest to users. By clustering links as such, they decrease users' costs of searching for what they want to read, making the user more likely to return to the website in the future. Because nearby links are close together in the tabular layout hierarchy of a web page, it is reasonable to assume that the hierarchical correlation assumption holds.

The reason why the correlation assumption is expected to hold for the URL feature is similar. The subject matter of an article is often correlated to its URL as evidenced by various news sites, such as CNN, that organize their stories into sections, which appear in the URLs for the stories. Also, the URL can convey even more information than merely the subject. For instance, if a page has a section with humorous articles, the humorous articles may have similar URLs, but assorted subject matter.

3.4 The Original Bayesian Tree Algorithm

This section describes the original tree learning algorithm that was developed by Shih [7]. Essentially, Shih's algorithm solves the problem of binary classification by computing, for each node x in the tree, the probability p_x^1 and p_x^0, the probability of the evidence in the subtree rooted at x given that x is of type 1 (has the trait) and of type 0 (lacks the trait). Let p^1 and p^0 represent the prior probabilities that the root is of types 1 and 0 respectively. The probability that a new node n has that trait can be computed using Bayes' Rule as the probability of the evidence underneath the root, $p^1 p_{root}^1 + p^0 p_{root}^0$, given that the new node has the trait, divided by the probability of the evidence in the tree

$$\Pr[n^1 \mid \text{all evidence}] = \frac{\Pr[n^1 \wedge \text{all evidence}]}{\Pr[\text{all evidence}]}. \tag{3.4}$$

Shih's algorithm computes each p_x^1 and p_x^0 by modeling the tree as a Bayesian network, so that

$$\begin{aligned} p_x^1 &= \prod_{y \in children(x)} M_{11} p_y^1 + M_{01} p_y^0 \\ p_x^0 &= \prod_{y \in children(x)} M_{00} p_y^0 + M_{10} p_y^1, \end{aligned}$$

where M_{ij} is the probability that the child's class is i given that the parent's class is j (M_{ij} can be thought of as probability of a mutation between generations in the tree that causes the child to be of class i when its parent is of class j). The leaf nodes, z, are the base case of this recursive definition; for example, p_z^1 is set to 1 if the leaf represents a positive sample and 0 if the leaf represents a negative sample.

In this thesis, this algorithm was generalized to handle an arbitrary number of classes (other than just 1 versus 0) by redefining the probabilities at each node to be

$$p_x^i = \prod_{y \in children(x)} \sum_{j \in classes} M_{ji} p_y^j, \tag{3.5}$$

for each class i.

One asset of this algorithm is that it yields an efficient, incremental update rule. When a new leaf x is incorporated into the tree, one way to update the p_z^i values for each node z in the tree is to recompute all of the values recursively as Equation 3.5 suggests. However, a key insight that leads to the fast incremental update algorithm is that most of the values of p_z^i in the tree remain the same after a new piece of evidence is added because none of their factors have changed, which follows from the fact that no evidence in their subtrees have changed. The only p_z^is that have changed lie on the path from the newly added leaf up to the root.

Each changed p_z^i will need to have exactly one factor changed, the factor corresponding the the child that has changed. These changes are made from the bottom up; the parent y of the leaf has each of its p_y^is modified to incorporate the new factor from the leaf, then the grandparent divides out the old values of p_y^i and multiplies in the new ones. Removing leaves can be accomplished analogously. For further details regarding how the incremental update rule is implemented, including pseudocode, see Section 7.4.1, which describes the implementation of the discrete Bayesian tree learner implemented for this thesis.

Chapter 4

The User Experience

This chapter describes the experience that a user has while configuring and using the recommender agent that was developed in this thesis.

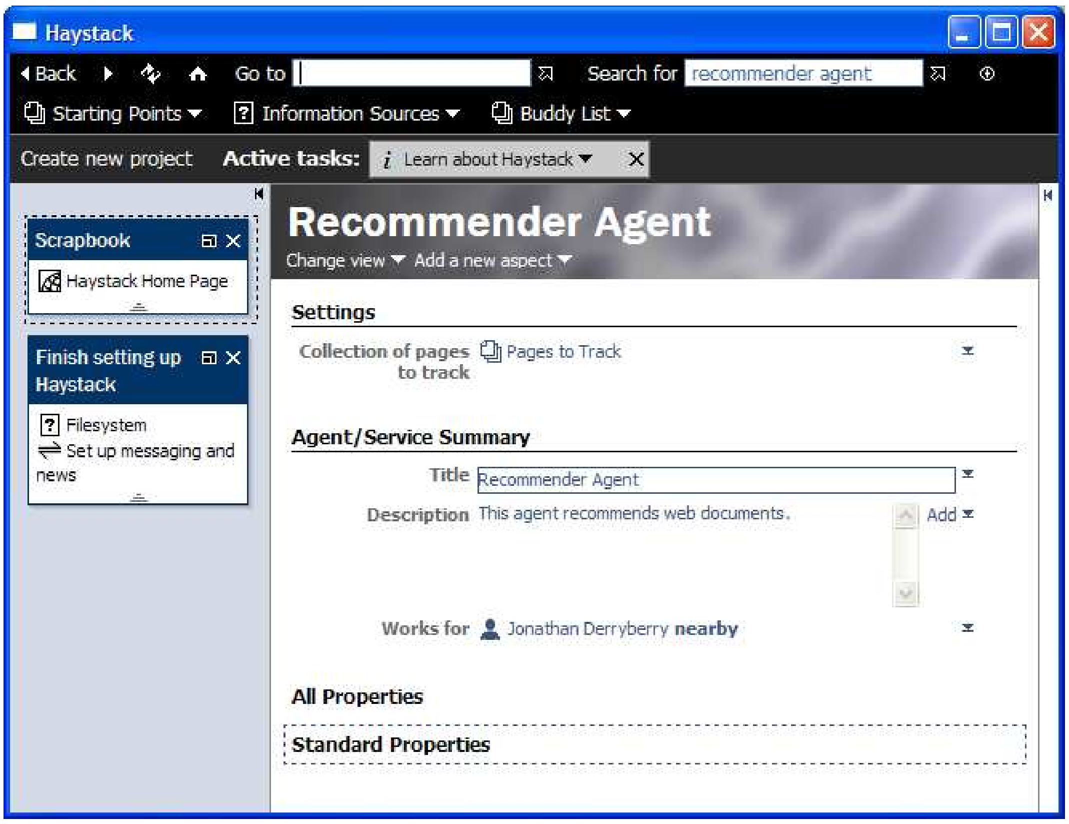
4.1 Configuring the Agent

When running Haystack, a user can open up the settings to the recommender agent by searching for "recommender agent." At this point, the user will be presented with the interface shown in Figure 4-1.

The one pertinent setting is the collection of pages to track. In order for users to be recommended web pages, they must provide the agent with a starting point or points to begin searches for links to new pages. The collection "Pages to Track" is just like any other collection in Haystack, web pages can be added to it via Haystack's drag and drop interface. As an example, Figure 4-2 shows a screenshot of the collection of pages to track after CNN's homepage has been dragged into it.

4.2 Using the Agent

Once the collection of pages to track has been formed, the recommender agent will begin recommending links that appear on those pages. The collection of recommended links is maintained by the recommender agent and updated periodically through-



Recommender Agent

Settings

Collection of pages to track: Pages to Track

Agent/Service Summary

Title: Recommender Agent

Description: This agent recommends web documents.

Works for: Jonathan Derryberry **nearby**

Figure 4-1: The agent setting interface, where the collection of pages to track can be accessed.

out the day. To view the recommendations, the user can search for the collection "Recommended Links," and keep the collection on one of the side bars, as shown in Figure 4-3.

Although the recommender agent will be able to make recommendations based on passive feedback obtained by listening to all of a user's visits to web pages, users also have the option of providing active feedback by categorizing web pages that they view. This functionality is shown in Figure 4-4.

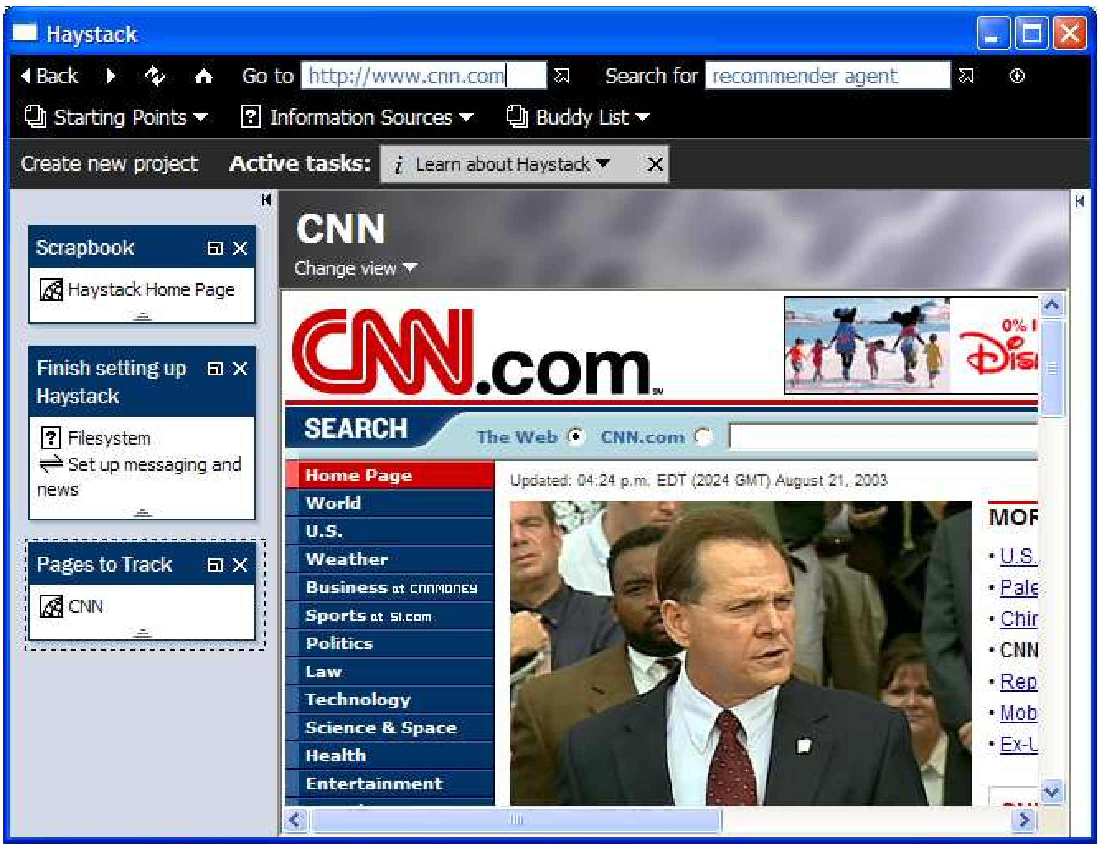
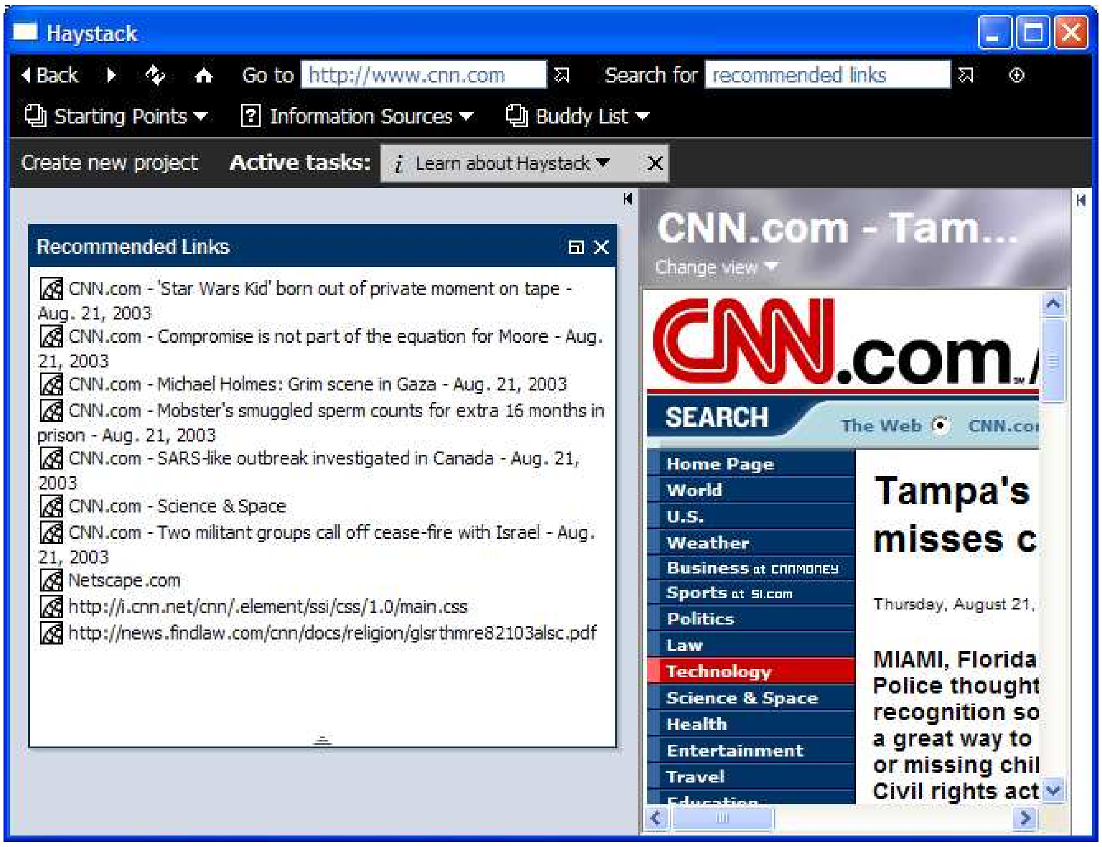
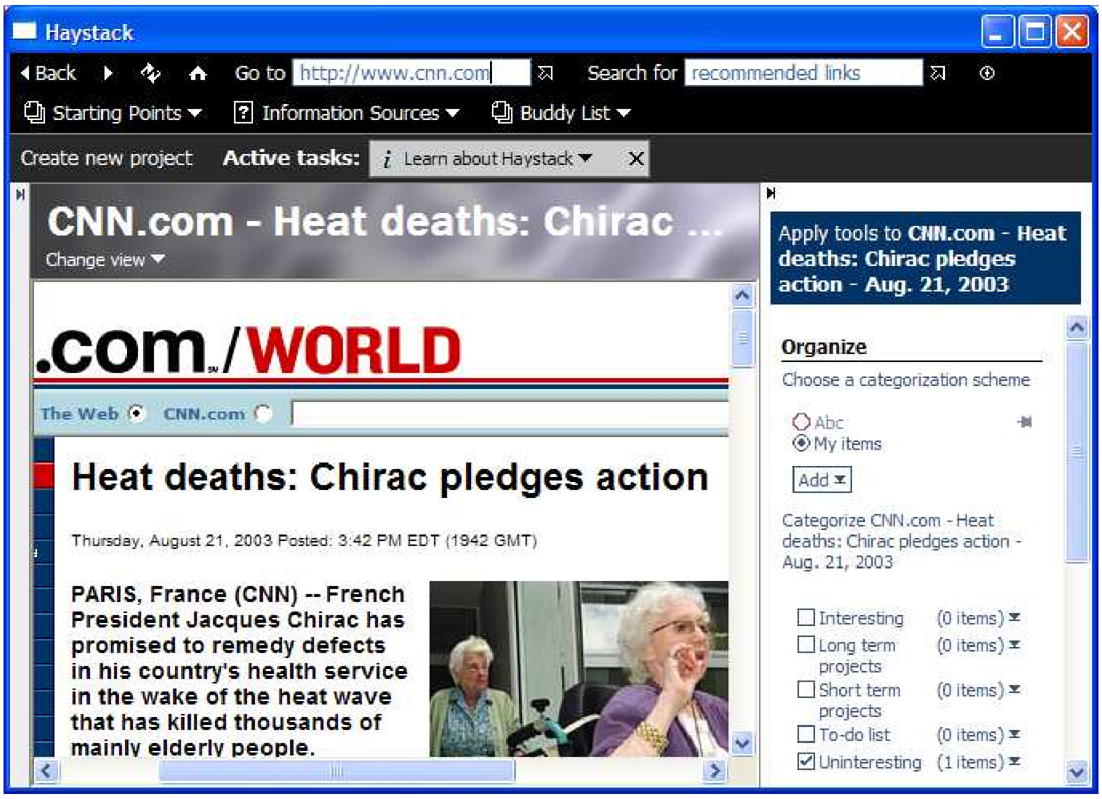


Figure 4-2: Web pages can be dragged and dropped into the collection of pages to track.



CNN.com - 'Star Wars Kid' born out of private moment on tape - Aug. 21, 2003
CNN.com - Compromise is not part of the equation for Moore - Aug. 21, 2003
CNN.com - Michael Holmes: Grim scene in Gaza - Aug. 21, 2003
CNN.com - Mobster's smuggled sperm counts for extra 16 months in prison - Aug. 21, 2003
CNN.com - SARS-like outbreak investigated in Canada - Aug. 21, 2003
CNN.com - Science & Space
CNN.com - Two militant groups call off cease-fire with Israel - Aug. 21, 2003
Netscape.com
http://i.cnn.net/cnn/.element/ssi/css/1.0/main.css
http://news.findlaw.com/cnn/docs/religion/glsrthmre82103alsc.pdf

Figure 4-3: The collection of recommended links, after visiting several pages on CNN.



Go to: http://www.cnn.com

Search for: recommended links

CNN.com - Heat deaths: Chirac ...

.com/WORLD

Heat deaths: Chirac pledges action

Thursday, August 21, 2003 Posted: 3:42 PM EDT (1942 GMT)

PARIS, France (CNN) -- French President Jacques Chirac has promised to remedy defects in his country's health service in the wake of the heat wave that has killed thousands of mainly elderly people.

Apply tools to CNN.com - Heat deaths: Chirac pledges action - Aug. 21, 2003

Choose a categorization scheme

Abc [✓ My items]

Categorize CNN.com - Heat deaths: Chirac pledges action - Aug. 21, 2003

☐ Interesting	(0 items)
☐ Long term projects	(0 items)
☐ Short term projects	(0 items)
☐ To-do list	(0 items)
☑ Uninteresting	(1 items)

Figure 4-4: An example of active feedback. Note that the page has been placed into the "Uninteresting" category.

Chapter 5

An Analysis of Tree Learning Algorithms

Before delving into the design and implementation of the recommender agent, the new tree learning algorithms that were developed for this thesis will be discussed in this chapter.

In the beginning of this chapter, a potential fault in the original discrete Bayesian algorithm (See Section 3.4) is identified and explained. Consequently, two alternative algorithms, one of which is a variant of the original, are proposed to rectify this weakness.

5.1 A Weakness in the Original Algorithm

The Bayesian tree learning algorithm has many good traits. It has a fast, incremental update procedure, and it seems to provide good performance in the setting in which only positive examples are used. However, there are some troubling aspects that could cause the algorithm to produce undesirable results.

First of all, if the user interaction model presented in Section 3.3 is accepted, then providing negative examples is important to producing good results. The intuition behind this claim is that without negative samples, a tree learner is likely to recommend the types of links that are visited most frequently as opposed to the types of links that are visited with the highest probability, such as rarely appearing links that are always visited on the rare occasions that they appear (e.g. a breaking news alert).

Second, if negative examples are used in the discrete Bayesian algorithm described in Section 3.4, then the ranking of the quality of a node in relation to others depends systematically on the number of nearby samples. To illuminate this problem, a toy example will be analyzed and then intuition for the problem will be presented.

Consider the example of a root node, r, that has C children, a fraction α of which are good (class 1) as supported by all of the data that has been gathered so far. If an unknown node x appears that is a child of r, we can compute the probability that it is good by using Equation 3.4 as follows

$$\Pr[x^1 \mid \text{all evidence}] = \frac{\Pr[\text{all evidence} \wedge x^1]}{\Pr[\text{all evidence}]}, \tag{5.1}$$

Where x^1 represents the event that new node x is of type 1. The probability of the evidence is given by the following formula

$$\begin{aligned}\Pr[\text{all evidence}] &= p^1 p_r^1 + p^0 p_r^0 \\ &= p^1 \prod_{y \in children(r)} (M_{11} p_y^1 + M_{01} p_y^0) + p^0 \prod_{y \in children(r)} (M_{00} p_y^0 + M_{10} p_y^1)\end{aligned}$$

Plugging the in the terms for the children, this becomes

$$\Pr[\text{all evidence}] = p^1 (M_{11}^{\alpha C} M_{01}^{C(1-\alpha)}) + p^0 (M_{10}^{\alpha C} M_{00}^{C(1-\alpha)}). \tag{5.2}$$

Now, under the reasonable and simplifying assumption that $p^1 = p^0 = \frac{1}{2}$ and that $M_{01} = M_{10}$, this reduces to

$$\Pr[\text{all evidence}] = \frac{1}{2}\left[(1-\beta)^{\alpha C} \beta^{C(1-\alpha)}\right] + \frac{1}{2}\left[\beta^{\alpha C} (1-\beta)^{C(1-\alpha)}\right], \tag{5.3}$$

where $\beta = M_{01} = M_{10}$. To find $\Pr[\text{all evidence} \wedge x^1]$, we simply need to multiply an extra factor into each of the terms, resulting in

$$\Pr[\text{all evidence} \wedge x^1] = \frac{1}{2}\left[(1-\beta)^{\alpha C+1}\beta^{C(1-\alpha)}\right] + \frac{1}{2}\left[\beta^{\alpha C+1}(1-\beta)^{C(1-\alpha)}\right], \quad (5.4)$$

which makes $\Pr[x^1 \mid \text{all evidence}]$ reduce to

$$\begin{aligned}\Pr[x^1 \mid \text{all evidence}] &= \frac{\frac{1}{2}\left[(1-\beta)^{\alpha C+1}\beta^{C(1-\alpha)}\right] + \frac{1}{2}\left[\beta^{\alpha C+1}(1-\beta)^{C(1-\alpha)}\right]}{\frac{1}{2}\left[(1-\beta)^{\alpha C}\beta^{C(1-\alpha)}\right] + \frac{1}{2}\left[\beta^{\alpha C}(1-\beta)^{C(1-\alpha)}\right]} \\ &= \frac{(1-\beta)^{\alpha C+1}\beta^{C(1-\alpha)} + \beta^{\alpha C+1}(1-\beta)^{C(1-\alpha)}}{(1-\beta)^{\alpha C}\beta^{C(1-\alpha)} + \beta^{\alpha C}(1-\beta)^{C(1-\alpha)}} \\ &= \frac{(1-\beta)\beta^{C(1-2\alpha)} + \beta(1-\beta)^{C(1-2\alpha)}}{\beta^{C(1-2\alpha)} + (1-\beta)^{C(1-2\alpha)}}.\end{aligned}$$

Now, we are ready to examine the effect of α and C on the recommedations that would be made by the algorithm. First of all, note that the only place α and C come into effect is in the exponents, by the value $C(1-2\alpha) = C(1-\alpha) - C\alpha$, which is simply the *difference* between the number of positive samples and negative samples, not their ratio. An example will illustrate this: let $C = 100$ and $\alpha = 0.3$, yielding

$$\begin{aligned}\Pr[x^1 \mid \text{all data}] &= \frac{\left[(1-\beta)\beta^{100(0.4)}\right] + \left[\beta(1-\beta)^{100(0.4)}\right]}{\left[\beta^{100(0.4)}\right] + \left[(1-\beta)^{100(0.4)})\right]} \\ &= \frac{[(1-\beta)\beta^{40}] + [\beta(1-\beta)^{40}]}{[\beta^{40}] + [(1-\beta)^{40}]}.\end{aligned}$$

Compare this to the case in which $C = 200$ and $\alpha = 0.4$

$$\begin{aligned}\Pr[x^1 \mid \text{all data}] &= \frac{\left[(1-\beta)\beta^{200(0.2)}\right] + \left[\beta(1-\beta)^{200(0.2)}\right]}{\left[\beta^{200(0.2)}\right] + \left[(1-\beta)^{200(0.2)})\right]} \\ &= \frac{[(1-\beta)\beta^{40}] + [\beta(1-\beta)^{40}]}{[\beta^{40}] + [(1-\beta)^{40}].}.\end{aligned}$$

In this example, the second case had a higher fraction of positive examples, but was declared equally likely to be good because it had more samples. With a little further inspection, it is clear that if the second case had more samples, it would be predicted as being *less* likely to contain a new positive node than the first case. In fact, in both cases, assuming that $(1-\beta)^{40}$ dominates the β^{40} term, the predicted probability that a new node is good is about β, the mutation probability, which could be far less than either 0.3 or 0.4. If $\beta = 0.1$, then the predicted probability that the new node is good would be about 0.1, which runs contrary to the intuition that it should be predicted to be good with probability 0.3 or 0.4.

Although this is only a toy example with a parent and one generation of children, it is clear that the underlying cause of the problem extends to the algorithm when applied to a whole tree. In fact, some thinking about the assumptions the algorithm makes about the underlying model sheds light on the root of the problem in this application: the binary probability space is incapable of capturing a phenomenon that is inherently continuous. In other words, the discrete Bayes algorithm tries to determine whether each node is “good” (i.e. has 90% good children) or “bad” (i.e. has 10% good children). There is no notion of “excellent,” “very good,” ”average,” and so on. However, to create an ordered list of recommendations that places excellent leaves ahead of very good leaves, it is necessary that the tree learner be able to make such a distinction because being very confident that a node is good is different from predicting that the node is excellent.

In order for the binary discrete tree learner to avoid this problem, the underlying process it is modeling must exhibit data that corresponds to its model, for example if the children of nodes in a tree were found to either be roughly 90% good or roughly 10% good, then the binary discrete tree learner would model the process well if the mutation probability were set at 10%. However, this appeared empirically not to be the case with web page recommendation.

5.2 Fixing the Weakness Using Buckets

Once this problem is identified with the discrete Bayes tree, one potential solution is to simply make the parameter space continuous, to reflect the true underlying process (e.g. clicks correlated to URLs or page layout). However, the problem with this is that it is a daunting task to model a continuous parameter space in a Bayesian net efficiently, as there are an infinite number of "classes" about which to store probabilities. However, making the complete transition to a continuous parameter space is unnecessary for most applications. One alternative is to discretize the parameter space into a finite number of *buckets*, which could represent, for instance, 0.5% good, 1.0% good, 1.5% good, et cetera, assuming that users will not care whether a document is 52% good or 52.5% good.

Using the generalized, multiclass version of the discrete Bayesian tree learning algorithm makes this transition easy; new classes are made to correspond to each bucket. The only thing left is to determine what the conditional probabilities of the children are given the parent nodes. A natural choice is to assume that the children's parameters are distributed according to a Beta distribution, centered about the parent's bucket.

The beta distribution is a good choice because it is assigns a positive weight to exactly the buckets on the interval $[0, 1]$ and is shaped like a bell curve centered about its expectation. Also, a prior can easily be placed on the distribution, if desired, to push the distribution of children slightly toward some a priori determined Beta distribution.

5.3 Developing a Tree Learner Intuitively

In addition to fixing the weakness using buckets, there are other alternatives for making a tree learner that is more amenable to solving problems with a continuous parameter space. This section describes an algorithm that was developed intuitively. Although it was not developed according to any probability model, it runs in linear time with a small constant, and directly addresses the problem with the discrete Bayes algorithm.

5.3.1 First Attempt at a New Tree Learning Algorithm

Let the vector $\vec{o}_x$ contain a histogram for the set of observations for some node x; for example, $\vec{o}_x = \langle 23, 44 \rangle$ if there were 23 positive examples and 44 negative examples contained in the set of leaves below x. Also, let $\vec{\rho}$ be a vector with values representing prior beliefs about the ratio of the distribution of the evidence, normalized so that its entries sum to 1; for example, $\vec{\rho}$ might equal $\langle 0.2, 0.8\rangle$ if each node in the tree was expected to have 20 percent positive examples. Now, define a constant α to be the weight given to ρ. This weight can be thought of as the the number of imaginary observations that ρ represents. Define the predicted distribution of the classes of new leaves below x to be

$$\vec{f_x} = \frac{\vec{o_j} + \alpha\vec{\rho}}{O_j + \alpha}, \tag{5.5}$$

where O_x represents the total number of samples for node x. For instance, continuing using the above example values, and if $\alpha = 30$, then Equation 5.5 becomes

$$\begin{aligned} \vec{f_x} &= \frac{\langle 23, 44\rangle + 30\langle 0.2, 0.8\rangle}{67 + 30} \\ &= \left\langle \frac{29}{97}, \frac{68}{97} \right\rangle. \end{aligned}$$

This provides a suitable solution if there are many observations, or if there are no known correlated observations from other sets. However, when learning over a hierarchy, it is natural to have nearby nodes, such as children and parents, affect our beliefs about the probability distribution $\vec{f_x}$, especially if there are few observations for x, but many observations for the parent, y, and some children, z, of node x. From our assumptions, data collected from a single node are assumed to come from the same distribution $\vec{f_x}$ and the distributions $\vec{f_y}$ and $\vec{f_z}$ of the parents and children are believed to correlate to $\vec{f_x}$ although there may be differences, which are expected to be slight, but possibly large.

Assuming that the severity of differences in distributions between neighbors is independent of their locations, $\vec{f_y}$ and $\vec{f_z}$ for each child z can be incorporated, with equal weight into $\vec{f_x}$ as follows

$$\vec{f_x} = \frac{\vec{o_x} + \alpha\vec{\rho} + \beta\vec{n}(x)}{O_x + \alpha + \beta}, \tag{5.6}$$

where $\vec{n}(x)$ is defined to be the average distribution of the neighbors of x, in other words

$$\vec{n}(x) = \frac{\vec{f_y} + \sum_{z \in children(x)} \vec{f_z}}{1 + |children(x)|}. \tag{5.7}$$

Although this definition for $\vec{f_x}$ is circular, $\vec{f_x}$ can be found for each node in the tree simultaneously because the equations for all of the $\vec{f_x}$s can be collected and solved as a system of equations.

5.3.2 Improving the Learner

Although the learner from the previous section seems to satisfy most of our intuition about how to learn over a hierarchy, there is a troubling caveat. Consider the following two scenarios:

- There is a root with little evidence but one child with much positive evidence
- There is a root with little evidence but many children, each with much positive evidence.

Intuitively, the second scenario should cause the root to be more positive than in the first because there are more child nodes supporting the hypothesis that it is positive. However, the above framework does not allow for this because all of the children are lumped together into one sum, which is normalized by the number of neighbors.

An alternative to the above algorithm is to lump the $\alpha\vec{\rho}$ term and the neighbors term ($\beta\vec{n}(x)$) together into one term that weights $\vec{f_x}$ toward its distribution. Intuitively, this model maintains the nice feature that real observations at each node can outweigh the effect of its neighbors and $\alpha\vec{\rho}$ term, but adds the feature that a higher number of neighbors diminishes the effect of the a priori determined $\alpha\vec{\rho}$ term. In this setting, the distribution that will weight $\vec{f_x}$ can be expressed as

$$\vec{\rho_x} = \frac{\beta\vec{\rho} + \sum_{y \in neighbors(x)} \vec{f_y}}{\beta + |neighbors(x)|}, \tag{5.8}$$

where $\vec{\rho_x}$ can be thought of as the expected value of the distribution at node x (igoring leaves beneath x), $\vec{\rho}$ can be thought of as the a priori determined expected value for the distribution of an arbitrary node in the tree, and β is the number of "virtual neighbors" that ρ represents. Then, $\vec{f_x}$ can be defined as

$$\vec{f_x} = \frac{\vec{o}_x + \alpha\vec{\rho}_x}{O_x + \alpha}, \tag{5.9}$$

which can be solved as a system of equations as the previous algorithm can.

5.3.3 Improving the Algorithm Again

One problem with the previous algorithm is that the distribution of a node is changed when children are added to it, even when they contain no evidence. For example, consider a parent node that has a fraction 0.7 of positive samples but, because of the weighting term, reaches an equilibrium with a value of 0.6 attached to the positive class. If a new node with no evidence were added as its child, then the parents's predicted fraction of positive samples may increase because the child will inherit its distribution from its parent, and the parent will perceive this as more evidence that its higher fraction of positives is correct. This does not make sense; although a new node has been added, no evidence has been added to the tree.

To be exact, there are two problems that can cause undesirable behavior from adding an empty node

1. The imaginary data from the weighting term affects neighboring nodes, even if the affecter is an empty leaf node.

2. Data from a single node is allowed to reflect back to itself as it propagates through the tree.

One solution to the first problem is to weight the average of the neighboring nodes so that nodes with little or no evidence have little or no influence on their neighbor's distributions. To present the solution, define the distribution of a node as usual to be

$$\vec{f_x} = \frac{\vec{o}_x + \alpha\vec{\rho}_x}{O_x + \alpha}, \tag{5.10}$$

with $\vec{\rho}_x$ redefined to be

$$\vec{\rho}_x = \frac{\beta\vec{\rho} + \sum_{y \in neighbors(x)} W_y \vec{f}_y}{\beta + \sum_{y \in neighbors(x)} W_y}, \tag{5.11}$$

where W_y is a weight term that ensures that neighbors with a small amount of data in their branches do not carry much weight when taking the weighted average of neighboring nodes.

A reasonable definition for W_y is the fraction of y's distribution that is based on real data, either in the node itself or from its neighbors. From Equations 5.10 and 5.11, W_y is thus defined as

$$W_y = 1 - \left(\frac{\alpha}{\alpha + O_y}\right)\left(\frac{\beta}{\beta + \sum_{z \in neighbors(y)} W_z}\right). \tag{5.12}$$

However, this strategy only solves the problem of imaginary data being regarded as real data if the weights for nodes whose branches contain no data are 0, and because data flows all around the tree, this may not be the case. Nevertheless, also incorporating a solution to the second problem from above will solve the first problem completely.

To prevent data reflection and solve the second problem listed above, a new set of distributions $\vec{h}_{x,y}$ is defined such that each $\vec{h}_{x,y}$ is the distribution that y sees coming from its neighbor x. The value of $\vec{h}_{x,y}$ is declared only to be influenced by nodes in y's branch to x (i.e. all nodes that y must reach via a visit to x). Each $\vec{h}_{x,y}$ can then be defined as

$$\vec{h}_{x,y} = \frac{\vec{o}_x + \alpha \frac{\sum_{z \in neighbors(x)-y} W_{z,x} \vec{h}_{z,x}}{\sum_{z \in neighbors(x)-y} W_{z,x}}}{O_x + \alpha}, \tag{5.13}$$

where $W_{z,x}$ is defined (similarly to W_y) as

$$W_{z,x} = 1 - \left(\frac{\alpha}{\alpha + O_z}\right)\left(\frac{\beta}{\beta + \sum_{y \in neighbors(z)-x} W_{y,z}}\right). \tag{5.14}$$

Finally, with the distribution of an arbitrary node x again defined as

$$\vec{f_x} = \frac{\vec{o}_x + \alpha\vec{\rho}_x}{O_x + \alpha}, \tag{5.15}$$

but with $\vec{\rho}_x$ redefined to be

$$\vec{\rho}_x = \frac{\beta\vec{\rho} + \sum_{y \in neighbors(x)} W_{y,x}\vec{h}_{y,x}}{\beta + \sum_{y \in neighbors(x)} W_{y,x}}, \tag{5.16}$$

both problems from above are solved.

A positive attribute of these definitions is that they yield an algorithm that can be solved in linear time; each $\vec{f_x}$ can be solved for explicitly and efficiently. Computing $W_{x,y}$ and $\vec{h}_{x,y}$ for all distinct $x, y \in T$ requires only $O(n)$ time because Equations 5.13 and 5.14 can be applied directly for each pair of neighbors in the tree if the tree is traversed in postorder first to compute the bottom up values and in preorder second to compute the top down values. Then, all of the knowledge exists to compute each $\vec{f_x}$ directly from Equations 5.15 and 5.16. Pseudocode for this algorithm is omitted for the sake of simplicity, although implementing the algorithm is straightforward from the equations.

5.3.4 Assessing the Algorithm

Although its genesis is based on intuition about the specific problems of learning about user interests over the hierarchy of URLs and tabular locations, and the algorithm may not be a good choice for tree learning in other contexts, the proposed tree learning algorithm has many positive attributes:

- No matter how many confident neighbors a node has, a significant number of observations at a node will outweigh the influence of the neighbors. This is because the aggregate effect of the neighbors is grouped into one term whose weight cannot exceed a constant number of real observations for the node.

- Branches of the tree that have no evidence do not affect the induced distributions of the other nodes in the tree.

- The computed distribution at evidenceless nodes that have many neighbors with the same distribution is closer to the distribution of the neighbors than if there were only one or two neighbors.

- The algorithm is efficient; it can be trained from scratch in linear time with a relatively small constant.

5.4 Weaknesses of the Tree Learning Algorithms

All of the tree learning algorithms mentioned in this thesis suffer from weaknesses resulting from the rigid tree structure that the tree learning framework imposes. All of them assume that the likelihood that a node is good can be completely encapsulated by the node's immediate neighbors. However, this is probably not true in general for either URLs or tabular locations.

In URLs, often times a parent will receive negative samples, but nevertheless have children with very positive distributions. For example suppose a user never clicks on any links to

`http://www.cnn.com/WORLD/<region>`,

but nevertheless often clicks on links of the form

`http://www.cnn.com/WORLD/<region>/<story>`.

Any of the above algorithms will deduce that a link of the form

`http://www.cnn.com/WORLD/<newregion>/<story>`

is bad because it is a direct descendant of a node that received a lot of negative evidence. There is no way to recognize that grandchildren of the `WORLD` node are usually good to recommend.

Another problem occurs when web editors place meaningless, dynamic information into URLs. For example, CNN places the date in their URLs, although this will probably not be a significant factor in whether or not a user wants to read the story.

This same problem occurs when using the layout of web pages. Home pages sometimes add or remove banners or other elements on their web pages, or move tables around in the page. Often much of the underlying structure of the layout remains the same. For instance, a headline news story may have location $[2, 3, 0, 0, 2]$ one day, but the next day an extra banner may be put into the page, making the layout position $[3, 3, 0, 0, 2]$.

To some extent, these problems can be solved heuristically, for example by deleting dates from URLs. However, an ideal tree learning algorithm would be able to deal with these problems gracefully without needing heuristics. One potential solution is to use a variant of nearest neighbors, defining the distance between two nodes to be the edit distance between two hierarchical descriptions. However, efficiently implementing such an algorithm would be a challenge and there would still be systematic problems with a straightforward nearest neighbors implementation (e.g. how to choose between equally distant neighbors).

Chapter 6

Design of the Tree Learning Package

The functionality of the tree learning package was based on the tree learning framework developed by Shih [7] (See Section 3.4). To maximize the utility of the tree learning architecture in this thesis, it was designed to be independent of the document recommender system as a whole. In fact, there is no reason why it cannot be used to learn about hierarchies of documents other than web pages, or even objects other than documents, such as photographs or animals, provided there is a hierarchical description of them.

Thus, the recommender system as a whole is merely one example of how to use the tree learning framework to learn about hierarchical data. At a high level, the tree learning architecture presents an interface in which the user of the learner adds evidence associated with a hierarchical description of an object to a tree learner. The user can also query nodes to determine the probability that the queried node is of a specified class.

6.1 Describing a Location in a Hierarchy

Locations of nodes in hierarchies can be described in diverse ways. Rather than use a string such as "`http://www.cnn.com/BUSINESS`," "200 Technology Square, Cambridge, MA U.S.A.," or "first table, third element, second subtable" to describe the location of a node in a tree, the hierarchical name of a node was abstracted using the `TreePath` class, provided by Java.

The constructor takes an array of objects representing a downward path, from the root toward the leaves, of a node in a hierarchy. The purpose of using the `TreePath` class as opposed to the URL directly is to make the tree learner flexible about what form the hierarchical description of a node may take; in other words, it provides a layer of abstraction between the hierarchy and how it is described. It is left up to the user of the tree learner to transform the features of their particular taxonomies into `TreePath` objects for use with the tree learning architecture. For example, users may change names such as "`http://www.cnn.com/BUSINESS`" and "200 Technology Square, Cambridge, MA U.S.A." into "[com, cnn, BUSINESS]" and "[U.S.A., MA, Cambridge, Technology Square, 200]."

6.2 The `IEvidence` Interface

Usually in tree learning or classification, the evidence objects added to nodes are merely labels corresponding to which class the nodes belong. However, the tree learning architecture was made more flexible so that the Bayesian algorithms could make use of their full power.

Each p_x^i in a Bayesian tree represents the probability of the evidence in the subtree rooted at x given that x is of type i. Thus, a piece of evidence in a tree need not be a simple label; it can have its own values of p_x^i defined. For example, suppose a fire engine is red with probability 0.9 and a sports car is red with probability 0.3. Then a leaf representing the evidence "is red" would have $p_{\text{is red}}^{\text{fire engine}} = 0.9$ and $p_{\text{is red}}^{\text{sports car}} = 0.3$.

To encapsulate the abstract notion of evidence, the `IEvidence` interface was written (See Figure 6-1). Its one method, `getProbability`, returns the probability that an object would exhibit the evidence if its class were `cls` (i.e. $p_{\text{the evidence}}^{\texttt{cls}}$), an object of type `LearningClass` (a class to abstractly represent the identity of the classes to which a node may belong).

```
public interface IEvidence {
    public double getProbability(LearningClass cls);
}
```

Figure 6-1: The `IEvidence` interface.

Also, for the specialized purpose of doing tree learning over two classes, the child interface `IBooleanEvidence` was created.

It should be noted that not all tree learning algorithms can easily make use of this generalized evidence framework. For example, the intuitive algorithm developed in Section 5.3 may produce undesirable results if used with evidence other than labels with variable confidence. For this reason, in hindsight, it may be desirable to change the `IEvidence` interface so that it only represents labels, with variable confidence.

6.3 The `ITreeLearner` Interface

The interface for tree learner objects is similar to other machine learning interfaces, containing methods for adding new training data and testing new objects. On the other hand, the tree learning framework is different from other machine learning interfaces because it requires that all objects used in training or testing have a hierarchical description, whether it be a URL or a taxonomic name of an organism. The `ITreeLearner` interface appears in Figure 6-2.

```
public interface ITreeLearner {
    public double getProbability(TreePath name, LearningClass cls);
    public void addEvidence(TreePath name, IEvidence ev);
    public void removeEvidence(TreePath name, IEvidence ev);
    public Iterator classes();
}
```

Figure 6-2: The `ITreeLearner` interface.

The `addEvidence` method associates the evidence `ev`, an object that implements the `IEvidence` interface, with the entity described by the `TreePath name`, which may contain something like the sequence of words "[com, cnn, LAW, petersoncase]." The `removeEvidence` method functions analogously. The `getProbability` method is used to find out the probability that an object with the description `name` is of the class `cls`, given all of the evidence in the tree. Because some implementations of the `ITreeLearner` interface do not update incrementally and need to be explicitly trained, `ITreeLearner` extends the `ITrainable` interface. The `ITrainable` interface is part of the generalized haystack learning framework, which is described in more detail in Chapter 8.

The `classes` method returns an `Iterator` over all of the classes about which the tree learner is learning.

Note that the `ITreeLearner` interface is flexible about what form the hierarchical description of an object may take and how many different classes of objects there are, although the underlying implementation may need to impose such constraints to be effectively implementable. However, these details are left up to the implementer.

6.4 The `ITreeLearningParameters` Interface

Although the use of an object implementing the `ITreeLearningParameters` interface is not required in an implementation of `ITreeLearner`, it has useful methods for constructing a Bayesian `ITreeLearner` similar to the two implemented in this thesis. The `ITreeLearningParameters` interface appears in Figure 6-3.

```
public interface ITreeLearningParameters {
    public double getPrior(LearningClass cls);
    public double getProbability(LearningClass parent, LearningClass child);
    public Iterator classes();
}
```

Figure 6-3: The `ITreeLearningParameters` interface.

The `getPrior` method returns the prior probability that a node is a member of the class `cls` (in Bayesian tree learners, the prior distribution of the root). The `getProbability` method returns the probability that the class of the child is `child` conditioned on the hypothesis that the parent's class is `parent`. The `classes` method returns an iterator over all classes described.

The implementations of these methods were hidden behind an interface to allow easy pluggability into the existing implementations of `ITreeLearner` should a better or different implementation of `ITreeLearningParameters` be developed.

Chapter 7

Implementation of the Tree Learning Package

This chapter describes the implementation of the tree learning package, including three implementations of the `ITreeLearner` interface described in Section 6.3. First, the tools that the tree learner implementations use are described, followed by sketches of how the tree learners themselves were built using the tools.

7.1 `BigSciNum` Prevents Magnitude Underflow

Initially, the implementation of the discrete Bayesian algorithm described in Section 3.4 used the `double` datatype to represent the probabilities (p_x^i) of each node in the tree. However, it soon became evident that the magnitude of the probabilities that were found in the tree quickly became so infinitesimal that a `double` would not suffice because the the smallest positive nonzero value of a `double` is 2^{-1074}, or about 4.9×10^{-324}. Because the probability of a group of evidence is the product of the individual probabilities, each piece of evidence can decrease the probability of the evidence at the root node by an order of magnitude. Thus, when more than a few hundred pieces of evidence are found, a `double` can no longer distinguish the probability of the root's evidence from zero. A few options were explored for solving this problem.

Using Java's `BigDecimal` class was considered, but there was a problem. A `BigDecimal` carries arbitrary precision; therefore, the amount of memory required to store all of the probabilities would be roughly $O(n^2)$, where n is the number of pieces of evidence added to the tree. This follows because there are $O(n)$ nodes, and after n pieces of evidence have been added to the tree, the number of bits in the probabilities of the nodes is $O(n)$ because there are $O(n)$ factors in the probabilities.

Another potential solution was to use the standard machine learning practice of using log-likelihoods. If instead of storing the probability of the evidence under each node, the logarithm of the probability of the evidence was stored, then the magnitude of the values stored would be unlikely to underflow the minimum positive value of a `double` because roughly 10^{324} pieces of evidence (more than the estimated number of elementary particles in the known universe) would have to be added to the tree if each piece of evidence decreased the likelihood of the evidence of the tree by a factor of 10. However, examination of the formula for the probability of the evidence at each node revealed that this method would not work. Consider the formula for the probability of evidence underneath node x given x is of type i

$$p_x^i = \prod_{y \in children(x)} \sum_{j \in classes} p_y^j \cdot \Pr[y^j \mid x^i], \tag{7.1}$$

where $children(x)$ is the set of children of node x and $classes$ is the set of all classes used. Now, to compute log-likelihoods, the formula changes to

$$\begin{aligned} \log p_x^i &= \log \left[\prod_{y \in children(x)} \sum_{j \in classes} p_y^j \cdot \Pr[y^j | x^i] \right] \\ &= \sum_{y \in children(x)} \log \left[\sum_{j \in classes} p_y^j \cdot \Pr[y^j | x^i] \right]. \end{aligned}$$

Thus, performing computations with the raw probabilities p_y^j is unavoidable because the terms of the second summation reside within a log[1].

[1]Near the end of this thesis, it was realized that log-likelihoods could be used. If $\log p_x^i$ is stored for each child of a node, then the parent y's values for each $\log p_y^i$ can be approximated by $\log p_y^i =$

To address the problems present in both of these potential solutions to the problem of precision underflow, the class `BigSciNum` was created. The `BigSciNum` class is analogous to Java's `BigInteger` and `BigDecimal` classes in that it is an immutable datatype representing a number, containing methods for all of the standard arithmetic operators. However, unlike `BigInteger` and `BigDecimal`, `BigSciNum` does not allow arbitrary precision. Instead, a `BigSciNum` object has a mantissa a, which is stored in a `double` and normalized to be in the interval $[1, 10)$, and an exponent b, which is stored in a `BigInteger` and can be any integer value. The value represented by a `BigSciNum` is $a \times 10^b$. The memory required by a `BigSciNum` m is a constant 64 bits for the mantissa plus $O(\log k)$ bits for the exponent, where k measures the magnitude of the number. Thus, the amount of memory consumed by the probabilities stored among all of the nodes is roughly $O(n \log n)$, where n is the number of evidence objects added to the tree.

It would have been reasonable also to use Java's 64-bit `long` datatype to store the exponent. This would have allowed slightly faster arithmetic operations on the exponent and allowed a slightly easier and more intuitive implementation. Nevertheless, the guarantee that the `BigInteger` class would never overflow outweighed the slight loss in performance of the addition and subtraction operations, even though about 10^{18} pieces of evidence would have had to have been added to cause a `long` to overflow.

7.2 Implementing `ITreeLearningParameters`

The most natural implementation of `ITreeLearningParameters`, which describes transition and prior probabilities in a tree learner (See Section 6.4), seemed to be a matrix. Thus, the `MatrixBasedParameters` class was created. The class's public method signatures are shown in Figure 7-1. The implementation of `MatrixBasedParameters`

$\sum_{x \in children(y)} \log \left[\sum_{j \in classes} 10^{\log p_x^j + \log \Pr[x^j|y^i]}\right]$, where $f = \log \left[\sum_{j \in classes} 10^{\log p_x^j + \log \Pr[x^j|y^i]}\right]$ can be approximately computed by finding $m = \max_{j \in classes} \log p_x^j + \log \Pr[x^j|y^i]$, and letting $f = m + \log \sum_{j \in classes} 10^{\log p_x^j + \log \Pr[x^j|y^i] - m}$, although precision may be lost because some values in the sum may suffer magnitude underflow.

```
public class MatrixBasedParameters implements ITreeLearningParameters {
    public MatrixBasedParameters(Matrix mutMatrix, LearningClass[] classes);
    public double getPrior(LearningClass cls);
    public double getProbability(LearningClass parent, LearningClass child);
    public Iterator classes();
}
```

Figure 7-1: The public methods and constructors of `MatrixBasedParameters`.

is essentially a wrapper for a custom, immutable `Matrix` class that was created. The `classes` argument to the constructor contains the identities of the classes represented by each row and column in `mutMatrix`. That is, entry i in `classes` represents the identity of the class whose entries in `mutMatrix` appear in both column i and row i. The entries of `mutMatrix` represent mutation probabilities. In particular, if node x is type i and node y is the child of x, then the probability that y is type j is stored in the entry at row j and column i in `mutMatrix`. For the convenience of the creator of a `MatrixBasedParameters`, the constructor normalizes the columns of `mutMatrix` so that they sum to 1.

From the entries of `mutMatrix`, `getProbability` has an obvious implemetation as does the `classes` method. However, to get the priors, matrix arithmetic is required. The most natural distribution of the prior probabilities for a node in a discrete Bayesian tree is a distribution such that each of the node's children would have the same distribution in the absence of evidence. This is the same as the class distribution of deep nodes in a tree that are far away from any evidence. Thus, the distribution of the priors is given by a vector $\mathbf{x}$ such that $\mathbf{Mx} = \mathbf{x}$, where $\mathbf{M}$ is the matrix containing the mutation probabilities. The column vector $\mathbf{x}$ is the steady state of $\mathbf{M}$. Its computation was placed in the `Matrix` class.

The `Matrix` class is an immutable implementation of a matrix with all of the basic matrix operators provided as well as convenience methods for making simple matrices. In addition, to facilitate the computation of priors, a `steadyState` method is provided that returns a column vector $\mathbf{x}$ such that $\mathbf{Mx} = \mathbf{x}$, where $\mathbf{M}$ is a $k \times k$ stochastic matrix for which the steady state is being computed. The `steadyState` method works by using the identity

$$\mathbf{Mx} - \mathbf{x} = \mathbf{0} = (\mathbf{M} - \mathbf{I})\mathbf{x}.$$

Then, the constraint that the entries of $\mathbf{x}$ sum to 1 is added by substituting $[1, 1, \ldots, 1]$ for the last row of $\mathbf{M} - \mathbf{I}$ and the column matrix $\mathbf{0}$ is changed to $[0, \ldots, 0, 1]'$. The reason why the substitution in $\mathbf{M} - \mathbf{I}$ can (and must) be performed is because $\mathbf{M}$ is stochastic and, therefore, underspecified because all of the columns sum to 1 so that any subset of $k - 1$ rows uniquely determines the other row. If the matrix computed by substituting in the last row is called $\mathbf{A}$, then the value of the steady state $\mathbf{x}$ is given by

$$\begin{aligned} \mathbf{Ax} &= [0, \ldots, 0, 1]' \\ \mathbf{x} &= \mathbf{A}^{-1}[0, \ldots, 0, 1]', \end{aligned}$$

where $\mathbf{A}^{-1}$ can be computed by Gaussian elimination.

Although the `MatrixBasedParameters` class is of the most direct utility to the discrete Bayesian algorithm, it is also useful for having a consistent set of priors between all of the tree learning algorithms. That is, a default set of parameters can be created, which all of the algorithms will use to determine the prior probability of the classes and the set of classes over which to learn, even though some of the algorithms may not explicitly use the mutation probabilities.

7.3 Implementing the `IEvidence` Interface

Two basic implementations of `IEvidence`, which tree learners use as training data, were written: `SimpleEvidence` and `CategoryEvidence` (See Section 6.2 for a discussion of the `IEvidence` interface). The class definition for `SimpleEvidence` appears in Figure 7-2. The `cls` argument to the constructor specifies the class that the evidence supports. The `confidence` parameter to the constructor is the same value returned by the `getProbability` method when queried on the class the evidence supports. For

```
public class SimpleEvidence implements IEvidence, Serializable {
    public SimpleEvidence(LearningClass cls, double confidence);
    public SimpleEvidence(LearningClass cls);
    public double getProbability(LearningClass nc);
}
```

Figure 7-2: The public methods and constructors of `SimpleEvidence`.

example, if `confidence` equals 0.9 (think the the evidence object as being the fact that the object looks like a pencil), then `getProbability` returns 0.9 when queried on "pencil" and 1 - 0.9 = 0.1 when queried on "eraser" or any other class of objects. Note that if there are more than 2 classes, it does not matter that the probabilities do not sum up to 1 because the probability returned is the probability of the evidence given the class, and not the other way around.

`CategoryEvidence` is similar to `SimpleEvidence`, except that it does not allow the confidence to be specified (it is assumed to be 1). Also, a `SimpleEvidence` object is only equal to itself, so if a node in a tree learner keeps a set of evidence associated with it, it will keep a copy of every `SimpleEvidence` object that has been added to it. On the other hand, `CategoryEvidence` objects are equal to any other `CategoryEvidence` that supports the same class. Therefore, when two pieces of `CategoryEvidence` are added to a set, only one copy of the `CategoryEvidence` is added. For example, if a leaf in a tree receives the `CategoryEvidence` "is interesting," and the user again categorizes the same object as interesting, only one `CategoryEvidence` object is added because they represent the same semantic idea; the fact that the described object is, in fact, a member of the specified category. On the other hand, there is no reason why evidence objects in general should have their identity defined by the class they support because in some tree learning applications it may make sense to add multiple identical but distinct `IEvidence` objects to the same object. `CategoryEvidence` was inspired by the haystack categorization scheme. When users of haystack categorize their documents, in no circumstance should two copies of the same categorization evidence be added, so the `CategoryEvidence` class was built to reflect this.

In addition to the basic `SimpleEvidence` and `CategoryEvidence` implementations, to implement the IBooleanEvidence interface, the analogous implementations BooleanSimpleEvidence and BooleanCategoryEvidence were written. Also, a VisitEvidence class, whose definition is in Figure 7-3 was built to capture a visit to a document. VisitEvidence is similar to its parent class BooleanSimpleEvidence,

```
public class VisitEvidence extends BooleanSimpleEvidence {
    public VisitEvidence(Date inVisitTime);
    public Calendar getTime();
}
```

Figure 7-3: The public methods and constructors of VisitEvidence.

but is hardcoded to support the "true" class (which was used to represent "interesting"). Also, VisitEvidence stores the date and time of access, and has its notion of equality based on the Calendar returned by the getTime method.

7.4 Implementing the ITreeLearner Interface

Three implementations of the ITreeLearner interface were created to reflect the three algorithms that were explored for tree learning. First, an implementation based on the discrete Bayesian algorithm was implemented. After the weakness was discovered in the discrete Bayesian algorithm (See Section 5.1), two alternative implementations were created based on the algorithms in Sections 5.2 and 5.3.

7.4.1 Implementing a Bayesian Tree Learner

To implement the ITreeLearner interface, first, the DiscreteBayesTree class was created (See Figure 7-4). This implementation of the ITreeLearner interface requires the user to pass in an ITreeLearningParameters to provide the prior and transition probabilities used for computing the probability of evidence at each node in the tree. The fxf parameter specifies the type of feature extractor that this tree learner will provide (See Section 8.2).

The mode parameter specifies how incremental changes to the tree should be made when addEvidence and removeEvidence are called, and it must have either the value

```
public class DiscreteBayesTree extends ATreeClassifier {
    public static final int RECOMPUTE;
    public static final int INCREMENTAL;
    public DiscreteBayesTree(ITreeLearningParameters p,
            IFeatureExtractorFactory fxf);
    public DiscreteBayesTree(ITreeLearningParameters p, int mode,
            IFeatureExtractorFactory fxf);
    public double getProbability(TreePath name, LearningClass nc);
    public void addEvidence(TreePath name, IEvidence ev);
    public void removeEvidence(TreePath name, IEvidence ev);
    public boolean isTrained();
    public void train();
    public Iterator classes();
}
```

Figure 7-4: The public methods and fields of the `DiscreteBayesTree` class.

`RECOMPUTE` or `INCREMENTAL`. The `RECOMPUTE` mode (which is still incremental, but performs extra recomputations to ensure there is no loss in precision from adding and removing evidence) should be used when the user is concerned about the loss in precision of the probabilities when many insertions and deletions are made in the tree, and is necessary if there is a risk of perfectly contradictory evidence, which occurs when two pieces of evidence, one of which may be a query, disagree about the value of a node yet both have 100% confidence in their assertions (if the fully incremental mode is used in this case, a divide by zero error occurs when a contradicter, typically query evidence, is removed). To see how `RECOMPUTE` works, recall the form of the probabilities at each node from Equation 7.1, which is replicated here for convenience:

$$p_x^i = \prod_{y \in children(x)} \sum_{j \in classes} p_y^j \cdot \Pr[y^j \mid x^i]. \tag{7.1}$$

Note that when $\vec{p_x}$ changes, the only values affected are $\{\vec{p_z} \mid z \text{ is an ancestor of } x\}$. Thus, it suffices to simply walk up the tree ancestor by ancestor and recompute $\vec{p_z}$ as suggested in Equation 7.1 and in the code in Figure 7-5, using only the cached values for the descendants along the way. Note that no imprecision results from repeatedly adding and removing the same evidence because no operation is ever undone; when evidence needs to be removed, it is not "divided out" but simply removed and all of the products containing changed factors are recomputed. To examine the time

```
private void handleChangeRecompute() {
    computeProbabilities();
    if (!isRoot()) {
        getParent().handleChangeRecompute();
    }
}
```

Figure 7-5: The private method called to update probabilities when in `RECOMPUTE` mode.

complexity of this method, assume that the number of pieces of evidence at each node is bounded by some constant. Then, the amount of work to compute the probabilities at each node is $O(bC)$, where b is the branching factor of the tree and C is the number of classes. Assuming that the number of classes is constant, this changes to $O(b)$. Because each ancestor needs to have its probabilities recomputed, the total time complexity to update probabilities in `RECOMPUTE` mode is $O(bh)$, where h is the height of the tree. If a constant branching factor is assumed and the height of the tree is $O(\log n)$, where n is the number of pieces of evidence added, then the total time complexity is $O(\log n)$, and the recompute update mode is efficient.

However, the above assumptions for b and h probably do not hold in the domains in this thesis, URLs and tabular layout positions. A more realistic assumption for URLs is that the *height* is constant. Then, by the fact that $b^h \approx n$, $b = \Omega(n^{1/h})$ so that the time complexity is at least polynomial in n. In fact, with the realistic assumption that the breadth is unbounded and the number of children of a particular node may comprise a constant fraction of the total number of nodes in a tree, the time complexity could be as much as *linear* in n. For some applications, this may be too slow.

To fix this problem, note that when $\vec{p_x}$ changes at some node x, the effect on its parent can be broken down into two steps. First, the initial value of $\vec{p_x}$ is "divided out" of its parent's product. Careful inspection of Equation 7.1 shows why this is possible. The value of p_z^j, where z is the parent of x, is simply a product of factors coming from z's children, with each factor independent of the others. Thus, $\vec{p_x}$ is divided out by computing

$$d = \sum_{i \in classes} p_x^i \cdot \Pr[x^i \mid z^j],$$

and dividing p_z^j by d. Then, z, whose value has just changed, is recursively divided out of its parent and so on. However, to get the recursion correct, note that to correctly divide out the parents from the tree, their old values must be used; hence, the recursion must start at the root and work downwards as suggested in Figure 7-6. Second, after $\vec{p_x}$ is updated to its new value, x is "multiplied in" to $\vec{p_z}$ using

```
private void divideOut(EvidenceNode changingChild) {
    if (!isRoot()) {
        getParent().divideOut(this);
    }
    // perform the actual dividing out for this node here
    ...
}
```

Figure 7-6: The private method called to divide out a node.

`multiplyIn`, an analogous procedure to `divideOut`. The main difference, besides the fact that each node on the path to the root is multiplied into the product of its parent rather than divided out, is that the child nodes are updated before the parents in the recursion so that each successive parent incorporates the new changes from its changed child. Compare the code in Figure 7-6 to the code in Figure 7-7.

```
private void multiplyIn(EvidenceNode changedChild) {
    // perform the actual multiplying in for this node here
    ...
    if (!isRoot()) {
        getParent().multiplyIn(this);
    }
}
```

Figure 7-7: The private method called to multiply in a node.

The `addEvidence` and `removeEvidence` methods operate as described above; the tree learner inserts or removes the evidence at the appropriate node, and immediately recomputes the cached probabilities as described above. The `getProbability` method simply returns the probability that the node with the specified `TreePath` belongs to the specified class. It works by first computing the probability of all of the evidence in the tree, according to

$$\Pr[\text{all evidence}] = \sum_{i \in classes} p^i p_r^i, \tag{7.2}$$

where r represents the root node, p^i represents the prior probability that the root is of class i, and p_r^i is the probability of the evidence in the subtree rooted at r (i.e. all of the evidence) given that r is of type i. Then it adds a new piece of evidence corresponding the the notion that the specified tree path is of the specified class, and recomputes the probability of the evidence in the tree, making use of the following formula

$$\Pr[x^i \mid \text{all evidence}] = \frac{\Pr[\text{all evidence} \wedge x^i]}{\Pr[\text{all evidence}]}. \tag{7.3}$$

Finally, note that the `train` and `isTrained` methods are irrelevant for this class because, regardless of the update mode, a `DiscreteBayesTree` is always ready to be queried immediately after `addEvidence` or `removeEvidence` is called. Therefore, `isTrained` simply returns `true` and `train` has an empty body.

7.4.2 Implementing a Continuous Tree Learner

Because of the problems discussed in Section 5.1, alternative implementations of the `ITreeLearner` interface were written. This subsection gives an overview of the implementation details regarding a bucket-based Bayesian tree, as described in Section 5.2.

The class `DiscretizedContinuousTree` implements the `ITreeLearner` interface using a multiclass `DiscreteBayesTree` underneath, which emulates a Bayesian tree with a continuous parameter space. An outline of the `DiscretizedContinuousTree` class appears in Figure 7-8. The basic, default constructor takes two arguments: an `ITreeLearningParameters` specifying the prior probabilities for the classes and an `IFeatureExtractorFactory` specifying the type of feature extractor to use (See Section 8.2). The `numBuckets` parameter to the second constructor specifies the number

```
public class DiscretizedContinuousTree extends ATreeClassifier {
    public DiscretizedContinuousTree(ITreeLearningParameters params,
            IFeatureExtractorFactory fxf);
    public DiscretizedContinuousTree(ITreeLearningParameters params,
            int numBuckets, int mode, IFeatureExtractorFactory fxf);
    public void add(FeatureList features, boolean cls);
    public double classify(FeatureList features);
    public boolean isTrained();
    public void train();
    public double getProbability(TreePath name, LearningClass nc);
    public void addEvidence(TreePath name, IEvidence ev);
    public Iterator classes();
    public void removeEvidence(TreePath name, IEvidence ev);
}
```

Figure 7-8: The class signature for `DiscretizedContinuousTree`.

of buckets into which the continuous interval $[0, 1]$ should be divided. The default value is 8, which seems to be the largest value that is practical for the recommender application. The `mode` parameter specifies the update mode that the underlying `DiscreteBayesTree` should use (See Section 7.4.1). The `ITreeLearningParameters` that is used for the internal `DiscreteBayesTree` is a `MatrixBasedParameters` object with the mutation matrix defined as described in Section 5.2 (with the distribution of child nodes of a particular bucket centered around the parent node according to a Beta distribution). The `add` and `classify` methods are from the general haystack learning framework. They override the implementations from `ATreeClassifier` because a `DiscretizedContinuousTree` can only handle binary evidence. See Chapter 8 for details about these methods. It should be noted that these details are not essential to an understanding of how to use the tree learning package in a context other than the general Haystack learning framework. The `isTrained` and `train` methods are from the `ITrainable` interface and have the same vacuous implementation contained in the `DiscreteBayesTree` class.

The `addEvidence` and `removeEvidence` implementations are not straight forward dispatches to the internal `DiscreteBayesTree`. There is a dichotomy between the internal nodes of the tree, and the evidence nodes. The internal nodes have many "bucket classes," while the leaf nodes containing the evidence only know of two classes, good and bad for example. Because internal tree nodes have a separate class for each bucket about which they learn, a reasonable interface between the internal nodes and the leaf nodes needs to be developed.

A natural way to use the internal tree follows when it is declared that a bucket represents the fraction α of its leaf children that are positive. Thus, when a positive leaf node appears, its parent node deduces, for each bucket α, that the positive evidence appears with probability α, and when a negative leaf node appears, its parent node deduces that the negative evidence appears with probability $1-\alpha$.

As an example, if there are 4 buckets, then the buckets will be centered at $\alpha = 0.125$, $\alpha = 0.375$, $\alpha = 0.625$, and $\alpha = 0.875$, and positive evidence will be made to return 0.125 if queried on the first bucket class, 0.375 if queried on the second bucket class, and so on. As a note, if the positive and negative probabilities of a leaf (p^+_{leaf} and p^-_{leaf}) are something other than 0 and 1 (e.g. 1 and 0.5), then its parent uses a linear combination of the those two probabilities ($\alpha p^+_{\text{leaf}} + (1-\alpha)p^-_{\text{leaf}}$). This gracefully handles the case in which the newly added evidence does not have full confidence, for example.

The `getProbability` method works by adding "test buckets" as leaves where the newly found object would fit into the tree, and computing the probability that the new object would fall into each bucket, as if it's quality were measured on a continuous scale rather than 0 versus 1. In this way the expected value of the new object's bucket can be found by taking weighted average of the αs, with weights determined by the probability that the new object falls into each bucket. In hindsight, it may have been simpler just to add binary test evidence to the tree and compute

$$\frac{\Pr[\text{new leaf is positive} \land \text{all evidence}]}{\Pr[\text{all evidence}]} \tag{7.4}$$

as in `DiscreteBayesTree`.

7.4.3 Implementing a Beta System Tree Learner

As an alternative to the DiscretizedContinuousTree class, another ITreeLearner implementation was developed according the discussion in Section 5.3. The class definition of BetaSystemTree appears in Figure 7-9. The params argument to both

```
public class BetaSystemTree extends ATreeClassifier {
    public static final double DEFAULT_ALPHA;
    public static final double DEFAULT_BETA;
    public BetaSystemTree(ITreeLearningParameters params,
           IFeatureExtractorFactory fxf);
    public BetaSystemTree(ITreeLearningParameters params,
        double alpha, double beta, IFeatureExtractorFactory fxf);
    public double getProbability(TreePath name, LearningClass cls);
    public void addEvidence(TreePath name, IEvidence ev);
    public void removeEvidence(TreePath name, IEvidence ev);
    public boolean isTrained();
    public void train();
    public Iterator classes();
}
```

Figure 7-9: The class signature for BetaSystemTree.

constructors specifies the parameters to use for the tree, but because mutation probabilities are not used by the algorithm, only the weights specified by the getPrior method and the classes method are used. The a and b parameters specify α and β respectively, which have the same definitions as discussed in Section 5.3. The fxf parameter specifies the type of feature extractor to use (See Section 8.2 for details about feature extractors).

After a BetaSystemTree is trained, the getProbability method needs only to look up the computed value for any queried node if it already exists in the tree, and has a simple formula for computing the correct probability to return if no corresponding node exists in the tree. The addEvidence and removeEvidence methods simply collect the evidence at the nodes in the tree without performing any other computations.

The isTrained method returns true exactly when there has been no evidence added to or removed from the tree since the last time train was called. The train method executes the final algorithm described in Section 5.3.

Chapter 8

The Haystack Learning Architecture

Haystack had an existing machine learning architecture that was developed by Mark Rosen in his Master's thesis [6]. However, the interfaces he developed were specific to textual learning algorithms or, more precisely, vector space learning algorithms. Because tree learning uses a hierarchical description of an object instead of a frequency vector, the existing machine learning architecture had to be generalized so that Haystack's applications would be able to use classifiers in a black box manner, regardless of the types of features the classifiers use.

When devising the general interface for machine learning classes in Haystack, one important consideration is that there are many disparate models and algorithms for learning, which use a variety of features. For example, vector space models use features such as the "bag of words" representation commonly used in text classification algorithms. On the other hand, a simple agent for filing documents according to author may simply use the "author" attribute of documents as its feature. Tree learning algorithms use a hierarchical description of an object as their only feature. Because of this disparity, it was determined that no top level interface for machine learning classes such as classifiers and categorizers could support universal methods for adding training data and classifying test data without giving instructions for how to get the features from an arbitrary `Resource` object. Therefore, the features corresponding to

Resource objects had to be abstractly represented.

8.1 The FeatureList Class

Before the feature extraction framework was finalized, the FeatureList class was created to encapsulate all of the features relevant to a particular algorithm (See Figure 8-1). Note that the only way a FeatureList can be created is by calling

```
public final class FeatureList implements Serializable {
    public FeatureList(Resource r, IFeatureExtractor ex);
    public int numFeatures();
    public Iterator iterator();
}
```

Figure 8-1: The FeatureList class.

its one constructor, which forces all FeatureList instances to be created by an IFeatureExtractor, which limits the possibility that a developer will mistakenly give an algorithm the wrong kind of features (See Section 8.2). Thus, developers are presented with a simple means of getting the features of an object without knowing anything about how they are gotten, or doing any preprocessing on the resource.

The FeatureList class is meant to be a wrapper for all of the features that an algorithm may find important for learning. For instance, a text learning algorithm may expect a FeatureList that contains one element, an IDocumentFrequencyIndex, while an ITreeLearner may expect a FeatureList that contains just a TreePath. A resource's features were made to be wrapped in a list as opposed to just providing a single feature (e.g. by creating a Feature class) because some learning algorithms may make use of more than one feature. For example, a composite algorithm for recommending web pages could make use both of an IDocumentFrequencyIndex for performing textual learning and a TreePath for performing tree learning.

8.2 The `IFeatureExtractor` Interface

The `IFeatureExtractor` interface, whose methods appear in Figure 8-2, was created in order to encapsulate a machine learning class's instructions for getting the features relevant to its learning algorithm. The `getFeatures` method returns an array

```
public interface IFeatureExtractor {
    public Object[] getFeatures(Resource r);
    public FeatureList getFeatureList(Resource r);
}
```

Figure 8-2: The `IFeatureExtractor` interface.

containing all of the features of a resource, and is used by the constructor of the `FeatureList` class (developers should not ever have to call this method explicitly). The `getFeatureList` method returns a `FeatureList` containing all of the features of a resource that are relevant to the feature extractor. Hence, each machine learning class should provide a corresponding `IFeatureExtractor` for getting the features it uses from a resource, given access to Haystack's data store. Examples of implementations of the `IFeatureExtractor` interface for tree learners appear in Sections 9.1.1 and 9.1.2.

In addition, an `IFeatureExtractorFactory` interface was created for factories that build `IFeatureExtractor` objects.

8.3 The `ITrainable` Interface

In order to capture machine learning in its most abstract sense, the `ITrainable` interface, which appears in Figure 8-3, was written. It contains three methods: `train` pre-

```
public interface ITrainable {
    public boolean isTrained();
    public void train();
    public IFeatureExtractor getSupportedFeatureExtractor(ServiceManager m);
}
```

Figure 8-3: The `ITrainable` interface.

pares the machine learning class for execution on test data, `isTrained` tests whether the machine learning algorithm is ready to perform its function (e.g. a classifier that was just trained), and `getSupportedFeatureExtractor` provides a developer with an abstract means of accessing the `IFeatureExtractor` that, given a resource, will extract a `FeatureList` containing the features that the particular `ITrainable` needs. The supplied `ServiceManager` provides a hook to Haystack's data store as well as a means of using other agents to help get the features.

An alternative to this scheme is to force machine learning algorithms to have general methods for adding data and testing data given a `Resource` object as a parameter, circumventing the need for feature extractors. However, this framework loses a level of abstraction by failing to decompose the tasks of learning and extracting the features. Also, machine learning algorithms do not need to know about which resources they are learning; they only need to know mappings from features to labels, and it is generally a good idea not to give classes more information than they need. If an algorithm needs to know which resources it is learning about, it should have its feature extractor include the resource object itself as a feature.

Thus, an example of using an `ITrainable`, whose underlying type need not be known, is as follows

1. Call `getSupportedFeatureExtractor` to get an `IFeatureExtractor`
2. Use this feature extractor in a black box manner to get features from the resources to be added as training data
3. Add the training data (via a subinterface of `ITrainable`)
4. Call the `train` method (if an untrained classifier is queried, the behavior is unspecified, and an exception may be thrown)
5. Use the feature extractor on the test resources to get the relevant features from the test data
6. Call testing methods on the extracted feature lists (e.g. call `classify`)

8.4 Creating General Classifier Interfaces

With the above framework in place, creating general interfaces for various types of classifiers (binary, multi-class, et cetera) is easy. The interface `IBinaryClassifier` appears in Figure 8-4 as an illustrative example of how the classifiers are structured. The `add` method adds training data having the specified feature list with the specified

```
public interface IBinaryClassifier extends ITrainable {
    public void add(FeatureList features, boolean cls);
    public double classify(FeatureList features);
}
```

Figure 8-4: The `IBinaryClassifier` interface.

class label, while the `classify` method returns a double representing the the most likely class (positive for true, negative for false) and whose magnitude measures the confidence the classifier has that its prediction is correct.

In addition to the `IBinaryClassifier` interface, the `IFuzzyBinaryClassifier` and `IMultiClassClassifier` interfaces were created. Discussion of them is omitted because of their similarity to the `IBinaryClassifier` interface.

Because of lack of need, methods for the removal of training samples were not included in these interfaces. Also, some of the existing learning interfaces in Haystack did not support the remove operation.

8.5 Incorporating Text Learning

Mark Rosen developed the existing textual learning framework for Haystack (See [6]), which has methods for adding labeled training data, where the feature parameter is an `IDocumentFrequencyIndex` (bag of words), and the class labels were `Objects`.

To incorporate Rosen's interfaces, the labels for data were all transformed to the typesafe `LearningClass` type. Additionally, because the interfaces already supported all of the operations of the general framework at an abstract level, it was apparent that no major changes to the existing code for the classes would need to be made.

Thus, instead of abandoning Rosen's text classifier interfaces, abstract text classifier classes were created. For example, the `ABinaryTextClassifier` class makes use of black box access to the methods of the `IBinaryTextClassifier` interface to implement the general haystack learning interface `IBinaryClassifier`. Then, existing `IBinaryTextClassifier` implementations, (e.g. `NaiveBayesBinaryClassifier`), were simply made to extend `ABinaryTextClassifier`, and consequently became implementers of the `IBinaryClassifier` interface. Similar abstract classes were made for the other text classification interfaces, such as `IFuzzyBinaryTextClassifier` and `IMultiClassTextClassifier`.

The only substantive method belonging to the `ABinaryTextClassifier` class was the `getSupportedFeatureExtractor` method; all other methods just call the obvious corresponding method in the `IBinaryTextClassifier` interface (e.g. the `add` method calls the `addDocument` method). The `getSupportedFeatureExtractor` method returns a `DFIExtractor`, which is a utility class that was written to be capable of returning an `IDocumentFrequencyIndex` corresponding to a specified `Resource`. A `DFIExtractor` works by connecting to other agents in Haystack that were designed for the purpose of extracting text and indexing the text into a bag of words representation (respectively, the text extraction agent and the Lucene agent).

8.6 Incorporating Tree Learning

The tree learning package, developed before the Haystack learning framework was generalized, was also incorporated into the framework. This was accomplished via the `ATreeClassifier` class, which appears in Figure 8-5.

Just as in the `ABinaryTextClassifier` class, the `ATreeClassifier` class makes use of black box calls to the methods of the `ITreeLearner` interface to implement the general interfaces. Thus, in order to implement the general interfaces, a tree learner needs only extend the `ATreeClassifier` class and implement the methods of the `ITreeLearner` interface.

Again the only substantive method belonging to the `ATreeClassifier` class was

```
public abstract class ATreeClassifier
            implements IFuzzyBinaryClassifier, ITreeLearner {
    protected ATreeClassifier(IFeatureExtractorFactory fxf);
    public void add(FeatureList features, boolean cls);
    public void add(FeatureList features, boolean cls, double confidence);
    public void add(FeatureList features, LearningClass cls);
    public void add(FeatureList features, LearningClass cls,
            double confidence);
    public double classify(FeatureList features);
    public MultiClassResult classifyMulti(FeatureList features);
    public IFeatureExtractor getSupportedFeatureExtractor(
            ServiceManager manager);
}
```

Figure 8-5: The `ATreeClassifier` class.

the `getSupportedFeatureExtractor` method; all other methods just call the obvious corresponding method in the `ITreeLearner` interface (e.g. the `add` method calls the `addEvidence` method). The `getSupportedFeatureExtractor` method returns an extractor that gets `TreePath` objects from `Resource` objects. There are currently two implementations of `IFeatureExtractor` pertinent to tree learners: one for getting a hierarchical description from a URL and one for getting a hierarchical description from the tabular location of links. These are described in Sections 9.1.1 and 9.1.2. The leaf name extractor to use is specified by the sole constructor of `ATreeClassifier`.

Chapter 9

Creating a Web Page Recommendation System For Haystack

This chapter describes how the driving application for this thesis, a web page recommendation system for Haystack, was built using the haystack learning framework (more specifically, the `IBinaryClassifier` interface), the existing functionality of Haystack, and a web spider capable of finding links on pages.

9.1 Extracting `TreePath`s from Resources

To use the `IBinaryClassifier` interface with an underlying `ITreeLearner` implementation, a factory class for `IFeatureExtractor` objects suitable to the tree learning interface must be created. The tree learning module uses hierarchical descriptions of objects, so the underlying features that will be put in the feature list to be returned by the feature extractor should be objects of type `TreePath`.

The two hierarchical features used in this thesis are the URL and the tabular location of the link to the URL's content, so two classes, `ResourceTreePathExtractor` and `LayoutTreePathExtractor`, were created for getting `TreePath`s corresponding to a resource's URL and tabular location.

9.1.1 Implementing `ResourceTreePathExtractor`

Implementing the `ResourceTreePathExtractor` class for getting a hierarchical description of a `Resource` based on its URL was simple in that it did not require interaction with the rest of the Haystack system, like the `DFIExtractor` class, which relied on other agents in Haystack. However, a few nontrivial methods were needed to rearrange the information in URLs before constructing the `TreePath` to be returned. The methods of the `ResourceTreePathExtractor` class appear in Figure 9-1.

```
public class ResourceTreePathExtractor implements IFeatureExtractor {
    public static ResourceTreePathExtractor getInstance();
    public static Resource normalizeForNavigation(Resource r);
    public static TreePath normalizeForHierarchy(Resource r);
    public FeatureList getFeatureList(Resource r);
    public Object[] getFeatures(Resource r);
}
```

Figure 9-1: The `ResourceTreePathExtractor` class.

The singleton design pattern is used for this class because its instances do not carry any meaningful state, and the `getInstance` method returns the sole instance of a `ResourceTreePathExtractor`. The two static methods, `normalizeForNavigation` and `normalizeForHierarchy`, provide the important functionality.

The purpose of the `normalizeForNavigation` method is to map all URLs that represent the same web page to the same string. For instance, the two URLs

- `http://www.cnn.com/BUSINESS/index.html`
- `http://www.cnn.com/BUSINESS/`

both represent the same web page, so they must be regarded as the same when examining the history of visits to web pages; otherwise, the recommender agent may recommend pages that have already been visited or recommend the same page twice. To accomplish this task, the heuristic of removing suffixes such as "`/`" and "`/index.html`" performed well in practice.

One final detail about the `normalizeForNavigation` method is that it is specified to be stable in the sense that calling it multiple times, each time feeding in the output of the previous normalization, does not change its return value. More formally, `normalizeForNavigation` has the property that

$$\forall x \in U,\ f(x) = f(f(x)), \tag{9.1}$$

where f represents `normalizeForNavigation` and U is the set of URLs.

The second significant method, `normalizeForHierarchy`, also has an important stabilization property: calling `normalizeForNavigation` before passing in the resource to `normalizeForHierarchy` should not affect the returned `TreePath`. This property is stated formally as

$$\forall x \in U,\ f(x) = y \implies g(x) = g(y), \tag{9.2}$$

where g represents `normalizeForHierarchy`. In practice, this can be achieved by calling `normalizeForNavigation` at the beginning of the `normalizeForHierarchy` method.

Thus, the `normalizeForHierarchy` method first calls `normalizeForNavigation` so that all URLs that are heuristically defined to represent the same link are given the same hierarchical description. Then, the characters in the resource name that delimit names in the hierarchy, such as `'/'` and `'?'`, are replaced by `'/'`. Then, hierarchically meaningless prefixes, such as "`http://www.`," are erased.

Next, all components of the URLs that consist solely of numerical digits are removed. For instance, the string "`cnn.com/2003/LAW/06/04/fbi.watch.list.ap`" is changed to "`cnn.com/LAW/fbi.watch.list.ap`." Number removal was an ad hoc heuristic that was adopted because many sites archive their stories by date in addition to content, even though readers may not care whether a story came from yesterday or today. Also, in the particular case of CNN, "`/2003`" appears in an even more significant position than "`LAW`" so that a user's browsing behavior in 2003 becomes almost completely irrelevant to the stories that are recommended in 2004.

One final modification is performed on the name representing the host. Host names are organized with the most specific name first and are delimited by the `'.'`

character, as in "money.cnn.com." Thus, the host name is tokenized using '.' and its order is reversed. Finally, the path of the URL is tokenized using the '/' character and all of the tokens are passed to the TreePath constructor.

In summary, in the normalizeForHierarchy method, normalizeForNavigation is first called, then prefixes are removed, followed by numbers. Finally, the host is rearranged and the string is tokenized for transformation into a TreePath object. An example of the total transformation of a URL is that

"http://www.cnn.com/2003/LAW/06/04/fbi.watch.list.ap/index.html"

becomes the TreePath "[com, cnn, LAW, fbi.watch.list.ap]."

It should be noted that normalizing URLs like this causes a risk of collision. That is, two or more distinct URLs could map to the same normalized TreePath. However, in the tree learning framework, this is not a serious problem because two such URLs would likely be highly correlated in the tree anyway if they were forced to be distinct. This could have been avoided by placing the full original unaltered URL as the most specific name for the node. However, because name collision was not observed in practice, the unaltered URL was not used because it would have added an unnecessary extra node each time evidence was added.

The getFeatureList and getFeatures methods are implemented via straightforward calls to normalizeForHierarchy.

9.1.2 Implementing LayoutTreePathExtractor

Implementing the LayoutTreePathExtractor class (See Figure 9-2) for getting a hierarchical description of a Resource based on the tabular location of its links was more complicated logistically than extracting one based on the URL. While the URL of an object can be completely encapsulated by the Resource itself, it would be nearly impossible to find out where links to the Resource are located from the link's resource name itself. Therefore, LayoutTreePathExtractors must somehow be able to determine where to look for the links to the resources for which feature extraction is being performed.

```
public class LayoutTreePathExtractor implements IFeatureExtractor {
    public LayoutTreePathExtractor(ServiceManager manager);
    public LayoutTreePathExtractor(Resource[] starts);
    public Object[] getFeatures(Resource r);
    public FeatureList getFeatureList(Resource r);
}
```

Figure 9-2: The `LayoutTreePathExtractor` class.

To accomplish this, both constructors have parameters that provide starting points for where to look for links to the resources whose tabular locations need to be determined. The `starts` parameter specifies a list of pages to be used as such starting points, while the `manager` parameter provides a hook to Haystack's data store, where the collection of pages to track is stored.

Once the html for the pages to search for the links is obtained, Java's built in html parsing capability can be exploited. To extract the `TreePath`s, a current tree path is maintained while parsing the HTML. Each time a `HTML.Tag.TABLE` or `HTML.Tag.TD` start tag is found, the current tree path adds another hierarchical level, while end tags decrease the hierarchical level and increment the index of a counter at the new current level. Thus, if the sequence: `TABLE`, link1, end `TABLE`, `TABLE`, `TB`, link 2, link 3 occurs, then the links are given the following `TreePath`s

- link 1 is named `[0, 0]`
- link 2 is named `[1, 0, 0]`
- link 3 is named `[1, 0, 1]`.

Finally, so that distinct categorical evidence can be given to distinct resources that appear at the same tabular location across time, the resource name is appended to the end of the `TreePath`.

9.2 Getting a List of Links to Rank

In order to be able to recommend links, a list of links to newly found web pages must be compiled. In this thesis, this was accomplished by creating a web spider that starts from initial pages and searches the Internet using standard graph search algorithms. For example, it might start at the node `http://www.cnn.com`, notice its neighbors, which appear as links on the web page, and then explore some of its neighbors. To allow easy pluggability of different implementations of spiders into the recommender system, the `ISpider` interface was created (See Figure 9-3). The `search` method

```
public interface ISpider {
    public Resource[] search(Resource[] starts, IAppraiser appraiser);
}
```

Figure 9-3: The `ISpider` interface.

of the `ISpider` interface takes an array of resources to use as starting points in the search for newly posted links, and an `IAppraiser` (See Section 9.3) to help the spider prioritize the search, if it is using search algorithms similar to best-first.

A rudimentary implementation of a spider was built first. Using Jeff Heaton's article and code as a guide, the `SimpleSpider` class was built [3] (See Figure 9-4). A `SimpleSpider` performs a breadth first search starting at the URLs described by

```
public class SimpleSpider implements ISpider {
    public SimpleSpider(int maxDepth);
    public Resource[] search(Resource[] starts, IAppraiser appraiser);
}
```

Figure 9-4: The `SimpleSpider` class definition.

`starts`. It uses Java's built in HTML parsing capability, from the `HTMLEditorKit` class, to extract links from web pages. The `maxDepth` argument to the constructor specifies the depth of the breadth first search. It should be noted that on commercial news sites, such as CNN's, any depth larger than 0, which only allows examination of links on the start page, causes a large performance dropoff.

However, keeping in sight the goal of using the spider as part of a web page recommendation system, the user of a spider may want to see web pages that are a few nodes deep into the search graph of a `SimpleSpider`. Unfortunately, finding such deep pages using a `SimpleSpider` requires the expansion of many irrelevant nodes.

Users creating systems like web page recommenders presumably have some mechanism for ranking web pages explored in the spider's search of the web. If it is assumed that web pages that receive high scores are more likely to link to other web pages that receive high scores, then a greedy search algorithm that only explores nodes that receive high scores can search deep into the graph without wasting a lot of time expanding irrelevant nodes. This motivated the creation of the `GreedySpider` class (See Figure 9-5). The `appraiser` argument to the `GreedySpider` constructor

```
public class GreedySpider implements ISpider {
    public GreedySpider(int numExpand);
    public Resource[] search(Resource[] starts, IAppraiser appraiser);
}
```

Figure 9-5: The `GreedySpider` class definition.

is used to value each of the links that the `GreedySpider` encounters as it searches. The `numExpand` argument gives the user precise control over how many links are expanded. The `search` method of a `GreedySpider` maintains a queue of all links that have been seen, and expands the link that has the highest appraised value according to `appraiser` during each step of the search.

9.3 Ranking Links that Are Found

Once a list of links is found, the recommender agent still needs to determine which of the links to recommend. This motivated the creation of the `IAppraiser` interface (See Figure 9-6). The methods for adding and removing observations make

```
public interface IAppraiser extends Comparator {
    public void addObservations(Observation[] obs, ServiceManager manager);
    public void removeObservations(Observation[] obs);
    public void train(ServiceManager manager);
    public boolean isTrained();
    public double appraise(Resource r);
}
```

Figure 9-6: The `IAppraiser` interface.

use of the `Observation` class, which is a wrapper for a `Resource` and an associated

`IEvidence` object. The `addObservations` method also needs a hook to Haystack's `ServiceManager` so that implementations of the `IAppraiser` interface are able to make use of the feature extracting framework (See Section 8.2), which may need to use Haystack's data store, or other agents.

The `IAppraiser` interface is simpler than classifiers in that it need not make a decision about to which class an object belongs, but more complex than classifiers in that it is responsible for extracting features of the resources in its observations. Its `train` method needs access to the `ServiceManager` because the feature extractor interface requires access to it.

The `IAppraiser` interface extends the `Comparator` interface so that its instances can be used by Java's `Arrays` and `Collections` classes to sort lists of links in descending order of their appraised value via a simple call to their `sort` methods.

The `appraise` method returns a double corresponding to the value of the specified `Resource`. It should be noted that value is simply a monotonic function; there are no semantic implications to one resource's being declared "twice as good" as another resource by an `IAppraiser`.

The Haystack learning framework provides a straightforward way to implement the `IAppraiser` interface. Hence, the `Appraiser` class was built using an underlying `IBinaryClassifier`. The only significant detail about its implementation, which simply uses the `add` and `classify` methods of the underlying binary classifier, is that it retrains the binary classifier from scratch every time the `train` method is called, even though many of the algorithms support incremental update. There were a few reasons for this choice:

1. Not all `IBinaryClassifier` implementations support adding new data after the classifier has already been trained (as mentioned in Rosen's thesis [6])

2. Most of the algorithms that were used were fast enough to accomplish their tasks within an acceptable time frame, regardless of whether they were trained from scratch

3. This choice avoids serialization of the classifiers when Haystack is shut down,

so the underlying classifiers need not implement the Serializable interface.

9.4 Listening to Changes in Haystack's Database

In order to use Haystack's learning framework to create a service that recommends web pages to users, there must be a mechanism by which the learning components receive training data, such as the fact that a particular web page has been visited or the fact that a user placed a resource in a certain category. Fortunately, Haystack already has a mechanism in place for getting such information.

Listening to relevant changes to the database only requires creating the appropriate RDFListener objects and adding suitable patterns to which to listen. To encapsulate this, the listeners were abstracted away into their own module, the URLEvidenceReporter class (See Figure 9-7). The manager argument to the con-

```
public class URLEvidenceReporter {
    public URLEvidenceReporter(ServiceManager manager);
    public synchronized Report getReportAndClear(Resource[] resToSearch);
    public static class Report {
        public Observation[] getAdded();
        public Observation[] getRemoved();
    }
    public void startListeners();
    public void stopListeners();
}
```

Figure 9-7: The class definition of URLEvidenceReporter.

structor serves as an IRDFEventSource that broadcasts changes in the database to registered RDFListeners. A URLEvidenceReporter creates internal RDFListener objects that listen for the addition and removal of relevant RDF statements to and from the data store. The startListeners and stopListeners methods are used to start and stop the internal listeners. When the user calls startListeners, the URLEvidenceReporter begins to listen to changes, and will continue listening to changes until stopListeners is called.

The user of a URLEvidenceReporter does not directly hear about changes to the database. Instead, the user must call getObservationsAndClear, which returns a `URLEvidenceReporter.Report` object that has methods for getting the lists of `Observations` that have been added and removed from the Haystack data store. Such removals might result from decategorization of a URL (i.e. the user does not think the page is interesting anymore). In addition to returning these observations, the `getObservationsAndClear` method also clears its memory of any statements it has heard about. The reason why the statements must be stored and should not be immediately broadcasted back to the `IAppraiser` is that the `URLEvidenceReporter` visits the pages the user is tracking to randomly pick negative evidence to complement visit evidence that has been collected. If these web pages are immediately contacted each time a new URL is visited, then performance suffers unacceptably because each time a user navigates to a web page, the the negative evidence extractor contacts all of the pages that are currently being tracked.

Finally, one important practical detail of the `URLEvidenceReporter` implementation is that the `RDFListener` objects use a separate thread to add observations to the sets to add and remove. Before adding or removing an observation, its `Resource` is checked to ensure that it represents HTTP content. This is accomplished by attempting to connect to the URL that the `Resource` represents. The reason why a separate thread is used is that the methods of the `RDFListener` objects must be allowed to return quickly so as not to stall the data store, which would happen while the HTTP content test waits for a response.

9.5 Putting `RecommenderService` Together

To provide the top-level functionality of the recommender agent for Haystack, the `RecommenderService` class was created as an extension of the `GenericService` class. The main task in writing the `RecommenderService` class is to override the `init`, `cleanup`, `shutdown`, and `performScheduledTask` methods of the `GenericService` class. All of these method overrides had straightforward implementations.

The `init` method prepares the components of the recommender system for use when Haystack starts running. In the recommender service, it does the following:

- Loads the collection of pages to track, which is maintained through the Haystack user interface, from the Haystack data store
- Loads the collection that it uses to contain the recommended links from the Haystack data store
- Loads the serialized tools, such as the `IAppraiser` it uses to rank web pages
- Makes a new `URLEvidenceReporter` and starts its listeners
- Makes a new spider for performing searches for new links.

The spider that it creates is a `SimpleSpider` with depth 0. Although the greedy spider was more sophisticated, it was decided that the `SimpleSpider` class was ideal for its efficiency and because it yielded good results during experimentation with the agent.

The `cleanup` method prepares the `RecommenderService` to be shut down, before any of the other agents shut down. To accomplish this, it does the following

- Stops the `URLEvidenceReporter`'s listeners
- Incorporates one final report from the `URLEvidenceReporter` so that visitations that have occurred since the last time the agent was run are not lost.

The reason why these operations are put into the `cleanup` method instead of the `shutdown` method is that code run during the `shutdown` method has no guarantee that all of the components of Haystack are still operational. The other components of Haystack are needed so that the underlying machine learning algorithms can still access other agents as well as the data store to extract features from newly added or removed observations. After the `cleanup` method is run, there are no further tasks that need to be performed before Haystack shuts down, so the `shutdown` method is not overridden.

Finally, the `performScheduledTask` method is overridden and contains all of the steps necessary to update the list of recommended links. It performs the following tasks:

- Incorporates the latest report from the `URLEvidenceReporter`
- Trains the `IAppraiser` with the newly collected data added to it
- Uses the spider to find new links
- Ranks the newly found links using the `IAppraiser`
- Updates the collection of recommended links accordingly.

One detail regarding this method is that the newly recommended links have their titles extracted via a utility class `TitleExtractor` written for this thesis. By doing this, when the user browses the collection of recommended links (entitled "Recommended Links"), the user sees the titles of the recommended links rather than the sometimes-arcane URLs.

Chapter 10

Testing the Recommender System

This chapter summarizes the results of testing that was done to analyze the quality of web page recommendations made using various ensembles of machine learning classes from the Haystack learning framework. The results show that all of the tree learning classifiers provide a substantial increase in recommending performance above traditional textual analysis of the articles. The best classifier according to these results was a composite classifier created from two tree learners, one that used the tabular layout feature and one that used the URL feature. Adding text to the composite classifier did not seem to help performance.

10.1 Experimental Setup

These tests use data that was collected by Kai Shih during a user study he administered in order to test his similar application, The Daily You. The user study subjects were shown a sequence of 5 home pages from CNN and asked to click a check box beside each of the links they liked on each page.

Using this data, leave-one-out cross validation was performed by testing the accuracy of the `IAppraiser` at predicting which links the user would click on each of the 5 pages, using the data from the other 4 pages to train the appraiser. Just as the `URLEvidenceReporter` does, negative samples were randomly added to the set of observations in equal number to the positive samples that were added (adding all of the negative samples may have given the algorithms a slight advantage that they would not normally have). It was enforced, however, that no negative samples were added for links that had been clicked, just as in the recommender agent. Precision-recall curves were created from these results by examining the predicted ranking of each of the links that each user clicked on the test pages. Finally, precision and recall were averaged from the 5 curves for each user.

One final detail regarding these experiments is that there was missing text data for some of the links. This was a problem because the user-study data was about a year old and the missing data, which may not have been important for the initial user study's purposes, was not able to be recovered. No text was available for links that were redirects (incidentally, this is also an issue with the current implementation of the text extraction agent in Haystack). Also, no text was available for stories that CNN posted from the Associated Press, apparently because the stories were removed from CNN's website shortly after they were posted. Although having missing data is not good, a reasonable workaround was to ignore all data regarding links for which no text was extracted. It should be noted, however, that such exclusion of training and test data could potentially slightly bias the results for each classifier one way or the other even though in practice it was found that excluding the links from stories with no extracted text only slightly affected the precision-recall curves for all types of classifiers.

Criticism of the experimental setup is presented in Section 10.4.

10.2 Results

This section presents the data obtained from the experimental setup described in Section 10.1. Throughout this section, various combinations of classifiers are used in the experiments. To provide concise, consistent notation, a map of standardized names is used for each type of appraiser (See Table 10.1).

The composite classifiers were created by merging two or three binary classifiers. For example, a `DiscreteBayesTree` using the URL feature was combined with a

Naïve-T1	Naïve Bayes text classifier using body text feature (Rosen's)
Naïve-T2	Naïve Bayes text classifier using body text feature (Simple)
Discrete-U	`DiscreteBayesTree` using URL as the feature
Discrete-L	`DiscreteBayesTree` using layout as the feature
Discrete-UL	composite classifier using `DiscreteBayesTree`s with URL and layout features
Discrete-TUL	composite classifier using `DiscreteBayesTree`s with URL and layout features, plus a Naïve Bayes text classifier
Beta-U	`BetaSystemTree` using URL as the feature
Beta-L	`BetaSystemTree` using layout as the feature
Beta-UL	composite classifier using `BetaSystemTree`s with URL and layout features
Beta-TUL	composite classifier using `BetaSystemTree`s with URL and layout features, plus a Naïve Bayes text classifier
Buckets-U	`DiscretizedContinuousTree` using URL as the feature
Buckets-L	`DiscretizedContinuousTree` using layout as the feature
Buckets-UL	composite classifier using `DiscretizedContinuousTree`s with URL and layout features
Buckets-TUL	composite classifier using `DiscretizedContinuousTree`s with URL and layout features, plus a Naïve Bayes text classifier

Table 10.1: Short hand notation for various appraisers, which are described by the classifier upon which they are based.

`DiscreteBayesTree` using the layout feature. Rather than devise a more theoretically sound model for synthesizing the two classifiers, such as mixture of experts, the average of the two classifiers' outputs was taken to be the output of the composite classifier.

This was reasonable because all of the implementations of the tree learner interface (as well as the Naïve Bayes text classifier) had their outputs determined by an estimation of the probability that the tested resources were good or bad; their return values were not a geometric interpretation of the test data, such as the dot product of a feature vector with other vectors, as in a Rocchio or support vector machine classifier. Therefore, an output of 0.6 on a test sample had the same meaning (in this case an 80% probability of membership in the good class), regardless of the underlying classifier.

One notable detail is that two different Naïve Bayes text algorithms were used.

Mark Rosen's implementation was more sophisticated and appeared to perform better as a web page recommender so his implementation was used as an example of the best web page recommending performance that could be expected of a standard text classification algorithm.

However, because the values returned by his `classify` method are not directly correlated to the predicted probability that documents are good (they are some function of log-likelihood), it did not make sense to use his classifier in composite classifiers of the simple form used in this thesis. Therefore, a simple Naïve Bayes implementation was created so that its `classify` method returned a value that had the same semantic meaning as the tree learners' `classify` methods (i.e. the return value was 1 minus twice the predicted probability that the object was bad). The simple Naïve Bayes implementation was used in the composite classifiers that used text.

As indicated in Table 10.1, appraisers based on Rosen's classifier are denoted by Naïve-T1, while appraisers based on the simple Naïve Bayes implementation are denoted by Naïve-T2.

10.2.1 Using the URL as the Feature

To test the performance of various tree learning algorithms when using a web page's URL as its feature, the performance of Discrete-U versus Beta-U versus Buckets-U was tested. The performances of Naïve-T1 and Naïve-T2 were included as baselines.

The resulting precision-recall curves (See Figure 10-1) showed that all three tree learners significantly outperformed Naïve-T1 and Naïve-T2. Discrete-U and Beta-U yielded top ranked links that were about 2.5 times as likely to be declared interesting as an average article, compared with a factor of less than 1.5 for the top recommendations of Naïve-T1. Buckets-U did not perform as well as the other two tree learners.

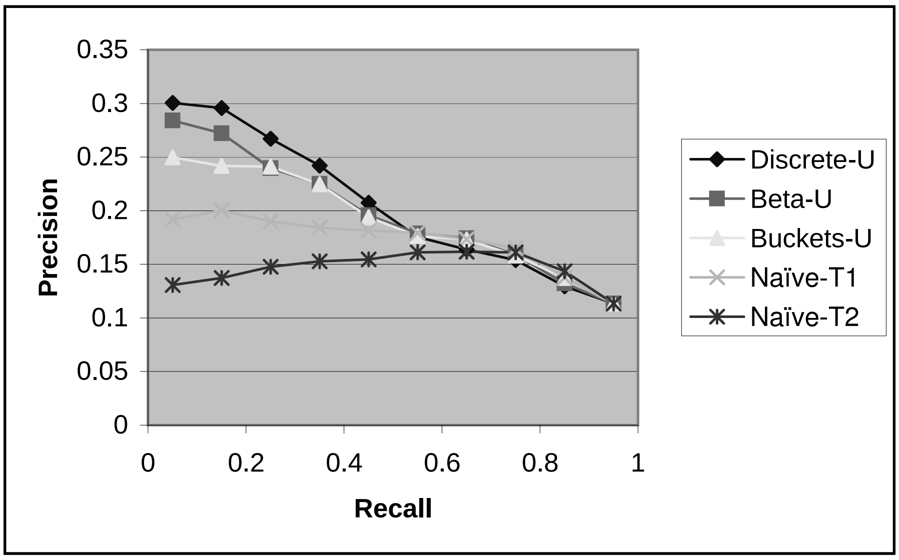


Figure 10-1: Precision-recall curves for Discrete-U, Beta-U, Buckets-U, Naïve-T1, and Naïve-T2.

10.2.2 Using the Layout as the Feature

To test the performance of various tree learning algorithms when using the tabular location of a web page's links as its features, the performance of Discrete-L versus Beta-L versus Buckets-L was tested. The performances of Naïve-T1 and Naïve-T2 were included as baselines.

The resulting precision-recall curves (See Figure 10-2) showed that all three tree learners significantly outperformed Naïve-T1 and Naïve-T2. Discrete-L and Beta-L yielded top ranked links that were over 2.5 times as likely to be declared interesting as an average article, compared with a factor of less than 1.5 for the top recommendations of Naïve-T1. Analogously to the previous section, Buckets-L did not perform as well as Discrete-L and Beta-L.

10.2.3 Using the Both the URL and Layout as Features

To test the performance of the composite classifiers that combined both hierarchical features, URL and the position of the links in the layout, the performance of Discrete-UL versus Beta-UL versus Buckets-UL was tested. The performances of Naïve-T1 and

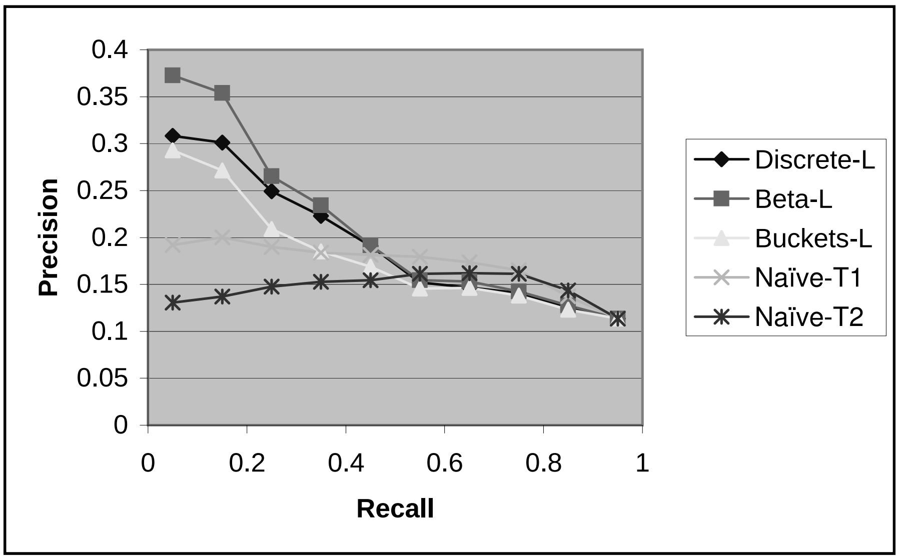


Figure 10-2: Precision-recall curves for Discrete-L, Beta-L, Buckets-L, Naïve-T1, and Naïve-T2.

Naïve-T2 were included as baselines.

The resulting precision-recall curves (See Figure 10-3) showed that all three tree learners significantly outperformed Naïve-T1 and Naïve-T2. Discrete-UL and Beta-UL yielded top ranked links that were about 2.5 times as likely to be declared interesting as an average article, compared with a factor of less than 1.5 for the top recommendations of Naïve-T1. Analogously to the previous sections, Buckets-UL did not perform as well as Discrete-UL and Beta-UL.

10.2.4 Using the Text, URL, and Layout as Features

To test whether adding textual analysis could improve the recommendations of some of the higher performing classifiers from the previous sections, Discrete-UL and Beta-UL were pitted against Discrete-TUL and Beta-TUL, with Naïve-T1 and Naïve-T2 again provided as baselines. Because the bucket-based tree learner had been slightly outperformed in previous experiments, it was omitted from the comparison performed in this section.

The resulting precision-recall curves (See Figure 10-4) showed that adding text to the simple composite classifiers that were created for this thesis did not provide a

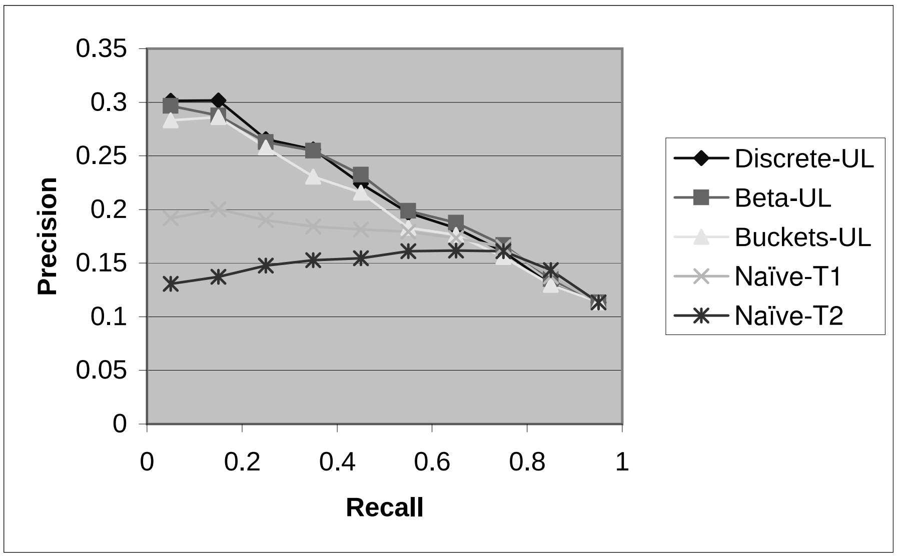


Figure 10-3: Precision-recall curves for Discrete-UL, Beta-UL, Buckets-UL, Naïve-T1, and Naïve-T2.

noticeable improvement in recommendation quality. Three suspected reasons for this are:

1. The text classifier that was used in the composite classifier was not as good as other text classifiers

2. The algorithm for composing the classifiers may be too simple to facilitate an incremental improvement from a text classifier

3. It is possible that text captures a similar subset of information to what the URL captures, the subjects of the articles, so that adding text to a composite classifier that already uses the URL is of little or no benefit.

10.2.5 Composite Versus URL-Only and Layout-Only

Comparisons were made among the successful types of classifiers:

- Discrete-UL was compared to Discrete-U and Discrete-L

- Beta-UL was compared to Beta-U and Beta-L

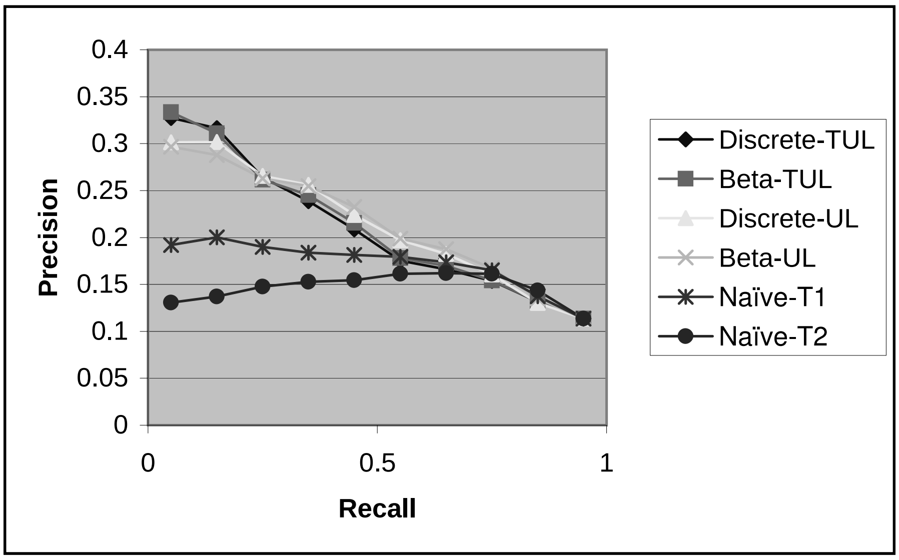


Figure 10-4: Precision-recall curves for Discrete-UL, Beta-UL, Discrete-TUL, Beta-TUL, Naïve-T1, and Naïve-T2.

Appraisers based on `DiscretizedContinuousTree` tree learners were not compared because of their slightly inferior performance.

The resulting precision-recall curves (See Figure 10-5) suggest that the composite classifiers may be slightly better than the highest performing single feature classifiers. While their performances were, for the most part, roughly equal on the low recall end of the curves, the composite classifiers offered more robust performance, with higher precision for intermediate recall values. However, all of the classifiers yield close enough performance to one another, that no decisive conclusion can be drawn.

10.2.6 Justification for Negative Samples

One of the benefits of tree learning algorithms is that they can make good predictions about which links a user will like without having any negative samples shown to them. However, in this thesis it is claimed that it is important to provide negative samples to tree learners in order to provide better recommendations. This claim motivates an experiment in which tree learners are compared with their identical counterparts with no random negative samples added.

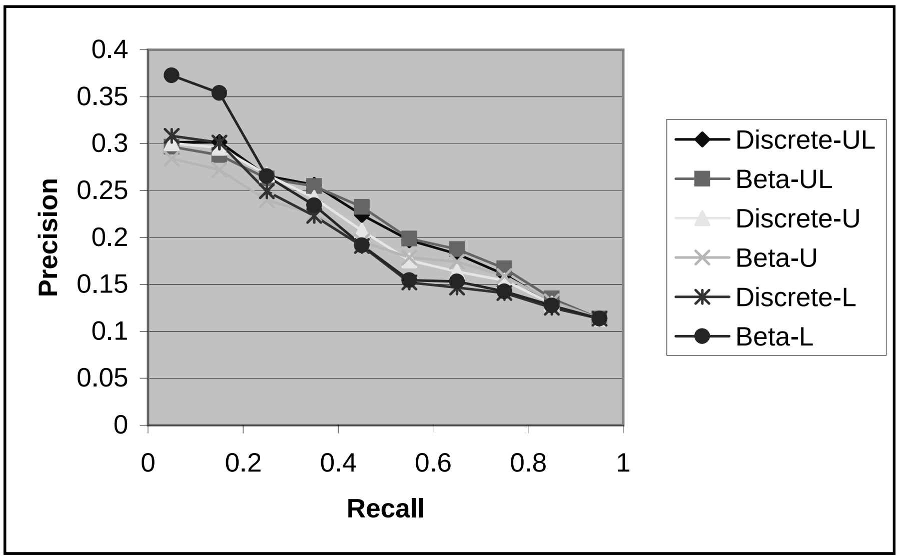


Figure 10-5: Precision-recall curves for Discrete-UL, Beta-UL, Discrete-U, Beta-U, Discrete-L, and Beta-L.

As a result, the precision-recall curves in Figures 10-6 were created for tree learners using the URL feature. The superior performance of the tree learners that used random negative samples suggests that it is imperative to provide tree learners with negative samples for optimal performance. However, Figure 10-7 shows that if the feature choice is changed to the layout feature, then the result is different; the tree learners with no negative samples perform just as well as the tree learners that received random negative samples. Although the results were mixed, the conclusion that was drawn from this experiment is that it appears to be safer to add negative samples than not to add them.

10.3 Summary of the Results

Between the three types of tree learners, the discrete Bayes and beta system tree learners seemed to provide the best performance, even though the discretized continuous tree learner was explicitly developed to solve a problem with the discrete Bayesian algorithm (See Section 5.1). Although this trend was surprising, it is possible that the number of buckets that were used, 8, was too low and unable to successfully model a

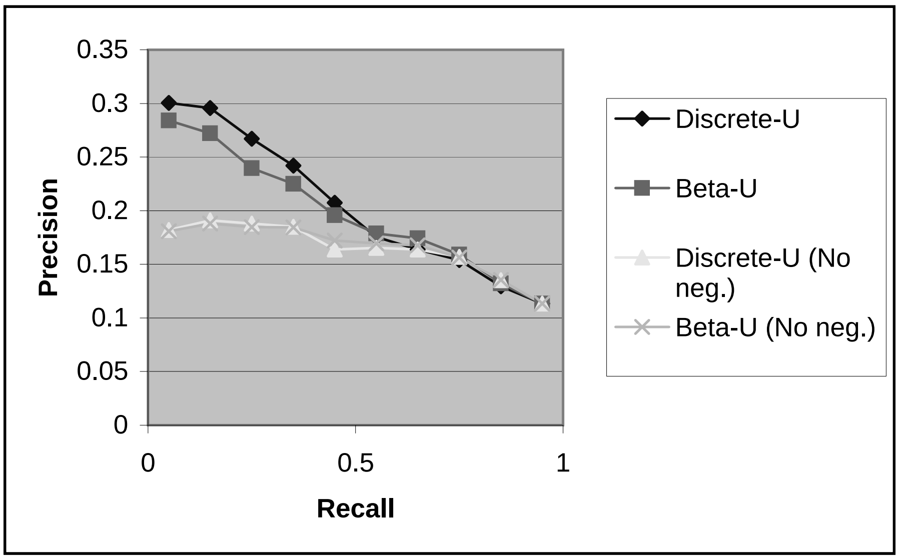


Figure 10-6: Precision-recall curves for Discrete-U and Beta-U compared against curves for the same algorithms with no negative samples added.

continuous distribution. The reason more buckets were not used was because learning and classification time scaled quadratically with the number of buckets that were used.

Also, it is possible that the bucket-based algorithm merely required more training samples for its performance to surpass the discrete Bayes tree learner. Indeed, the training data for each subject was too meager for precise trends in browsing behavior to be identified because there were often as few as a dozen positive training samples. Thus, it was expected that the systematic weakness of the discrete Bayes algorithm would not show up in this experiment. Also, perhaps the text-based appraiser would have been more competitive with the tree learners if more training data had been provided.

Among the various feature sets that were used, the combination of URL and layout position seemed to yield the best performance. One reason for this is that even though URL and layout position are often correlated, (e.g. similar subjects are placed in similar areas), layout position can offer additional information about a story besides its subject, such as whether it is the front page news story or another headline.

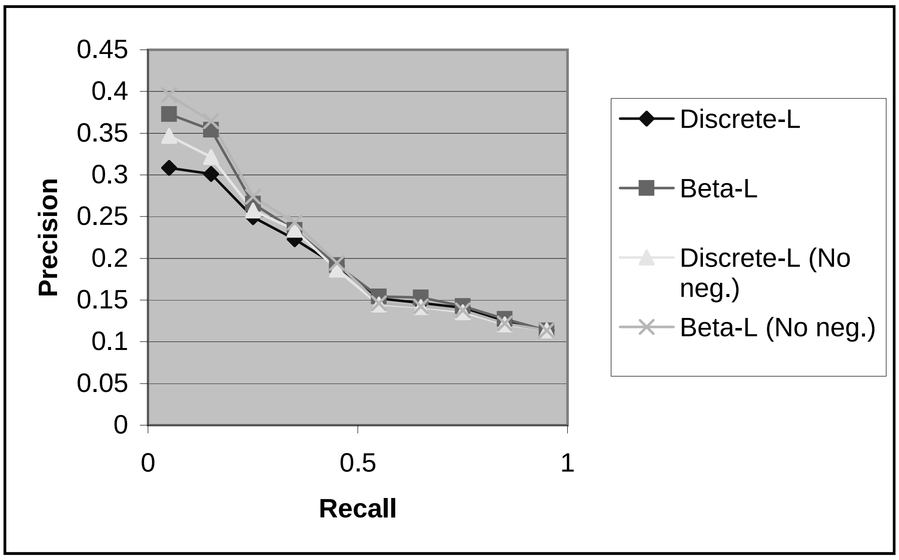


Figure 10-7: Precision-recall curves for the Discrete-L and Beta-L algorithms compared with curves for the same learners with no negative samples added.

With more training data, and a smarter way of synthesizing multiple classifiers than just the arithmetic mean, adding textual learning to the top performing classifiers would be expected to improve recommendations. However, these experiments show that without such enhancements, textual learning at best provides little incremental value to a recommendation system built on tree learning over URLs and layout positions.

10.4 Criticism of the Experimental Setup

Although the test results were encouraging, and experimentation with the recommender agent itself suggests that the system works well, there are faults to the experimental setup that warrant a brief discussion.

First, the training data from the experimental setup is purer than the training data that the agent would get in the real-world setup because, among other things, the agent is unable to distinguish between visitations that are made as an intermediate step toward locating a story to read and the end destination of a user's navigation to a story. In the user study, users explicitly select the stories that they would have read.

Also, only CNN's web page was used; other web pages may have different formats, which may affect the quality of the recommendations. For example, many of foxnews.com's stories do not exhibit an organized hierarchical URL format, so only tabular format is likely to be useful for recommendations at such sites.

Additionally, because the user study was only carried out in the short run, over 5 home pages, the tests are unable to show how different classifiers will perform in the long run. Some classifiers could increase in accuracy, while other's may begin showing systematic problems when they receive a lot of data (See Section 5.1).

Nevertheless, the trends observed in the test results are sufficiently marked that they succeed in demonstrating the macroscopic result that using the URL and tabular location of links appears to be essential to providing the most accurate recommendations to the user.

Chapter 11

Conclusion

As a result of this thesis, a working web page recommending agent was created for Haystack. The choices for the algorithms and agent parameters were made so that its existence will be transparent to users who choose not to use it. On the other hand, it provides a simple interface and reasonably good recommendations so that someone who chooses to use the agent can benefit from its recommendations with minimal effort overhead.

The two tree learning algorithms that were developed in this thesis also showed promise, with `BetaSystemTree` offering similar performance to the discrete Bayes algorithm, but solving the potential long run problem identified in this thesis.

11.1 Future Work

Although this thesis describes the design and implementation of a working application for Haystack, there are many ways in which the work in this thesis could be continued and improved.

11.1.1 Tree Learning

Tree learning could be improved with the creation of new algorithms that reflect the underlying correlation more accurately than the current tree learning algorithms.

Although the correlation of URLs and layout positions to user interest appear to be remarkably strong as evidenced by the success of the tree learners in the experimental trials of this thesis, the tree learners could be made more robust by addressing the following issues

- Sometimes there are dynamic hierarchical elements, such as the date's being placed in a URL, that bear little or no correlation to the user's interest in the page

- In tabular layouts, sometimes top level banners are added to the page, or the tables are moved around at a high level while all of the nesting remains unchanged.

Gracefully dealing with these issues in a theoretical manner, rather than heuristically would likely provide a more robust classifier. One potential solution may be to use some form of nearest neighbors, using string edit distance as the distance metric, where each level in the taxonomy is defined to be a "character." Another potential solution is to model URLs and tabular layout positions with a Hidden Markov Model.

11.1.2 The Haystack Learning Framework

Although the general structure for the Haystack learning framework is reasonable, there are many interfaces that could be added to it to allow users to leverage its implementers' power more fully.

For example, a subinterface for classifiers that support the removal of data could be added. This would have made learning new classifiers from scratch unnecessary in the context of the `Appraiser` class.

Additionally, an interface for machine learning classes that support incremental updates would be helpful because it would allow its users to guarantee that the incremental classifiers would not need retraining after data was added to or removed from them.

Finally, an interface could be created for probabilistic classifiers that can return an estimation of the probability that the tested resource is of a particular class (i.e. classes that support methods similar to `ITreeLearner`'s `getProbability` method).

11.1.3 Recommender Agent

There are several modifications that may increase the utility of the recommender agent to the Haystack system.

- A better method for combining the binary classifiers than using the arithmetic mean would likely improve the performance of recommendations based on composite binary classifiers.
- There may be additional features other than text, URL, and layout position that could increase the accuracy of recommendations.
- It may be possible to improve evidence reporting. Currently, the technique for adding negative samples is fairly ad hoc because it randomly chooses negative samples from all of the pages that are being tracked.
- It may be desirable to provide a separate list of recommendations for each page that is being tracked so that users can have some control over their view of the top recommendations.
- In order to handle situations in which a user's tastes change, it may be beneficial to decay the confidence of training data as it ages.
- Providing stronger negative feedback from recommended links that are ignored, and weaker positive evidence for recommended links that are visited may be desirable because such links receive more exposure to users so that they may be more likely to be clicked, all other things being equal.

11.2 Contributions

Lastly, a list of the major contributions made throughout the course of this thesis is presented:

1. Criticism of the existing discrete Bayes tree learning algorithm, including the identification of a potential systematic long run flaw

2. Proposition of two algorithms that solve the above flaw, a bucket-based emulation of a Bayesian network with a continuous parameter space and an intuitively developed algorithm (as implemented in `BetaSystemTree`).

3. Implementation of a tree learning package for Haystack including implementations of 3 tree learning algorithms.

4. Development of a general Haystack learning framework.

5. Integration of both the existing text learning framework and the newly created tree learning package into the general Haystack learning framework.

6. Development of a modular, extensible, and lightweight web page recommending agent for Haystack.

7. Demonstration of the capability of the agent via testing.

Bibliography

[1] Corin R. Anderson and Eric Horvitz. Web montage: a dynamic personalized start page. In *Proceedings of the eleventh international conference on World Wide Web*, pages 704–712. ACM Press, 2002.

[2] D. Billsus and M. Pazzani. A hybrid user model for news story classification, 1999.

[3] Jeff Heaton. Programming a spider in java. Technical report, developer.com, 2003.

[4] D. Heckerman. A tutorial on learning with bayesian networks, 1995.

[5] Ken Lang. NewsWeeder: learning to filter netnews. In *Proceedings of the 12th International Conference on Machine Learning*, pages 331–339. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA, 1995.

[6] Mark Rosen. E-mail classification in the haystack framework. Master's thesis, Massachusetts Institute of Technology, 2003.

[7] Lawrence Shih and David Karger. Learning classes correlated to a hierarchy, 2003.